IBM Personal Systems Developer

A Publication of the IBM Developer Assistance Program
Spring 1990

- OS/2 2.0
- Spotlight on PageMaker
- OS/2 Performance Tuning

IBM Personal Systems Developer

Table of Contents

Editor's Comments 3

Developer Assistance Program
Developer Assistance Program Update 4

Spotlight
PageMaker 3.0 for OS/2 6

32-Bit OS/2
OS/2 2.0 Overview 16
OS/2 2.0 32-Bit API 23
OS/2 2.0 Tools and Program Development 31
Multiple Virtual DOS Machines: A Better DOS 36
OS/2 Numeric Support 45

OS/2 Standard Edition
Performance Tips and Guidelines 50
Writing OS/2 Device Drivers in C 61

Extended Edition Communications Manager
Optimizing APPC Performance Over LANs 67

Software Tools
Tool Selection Techniques 75
MKS Toolkit for OS/2 79
Applications Manager for PM Development 83
Powerful Voice Processing Under OS/2 89

International
OS/2: A Worldwide Product 95
OS/2 National Language Support 96
National Language Support for the Dialog Manager 102

Extended Edition Database Manager
Query Manager Customized Interface 105
Database Configuration Parameters 114

The Market
Market Update 121

Resources
OS/2 Books 126

© Copyright 1990 by International Business Machines Corporation. Printed in U.S.A.

IBM Personal Systems Developer is published by the Entry Systems Division of International Business Machines Corporation, Boca Raton, Florida, U.S.A., Dick Conklin, Editor.

Titles and abstracts, but no other portions, of information in this publication may be copied and distributed by computer-based and other information service systems. Permission to republish information from this publication in any other publication or computer-based information system must be obtained from the Editor.

IBM believes the statements contained herein are accurate as of the date of publication of this document. **However, IBM hereby disclaims all warranties either expressed or implied, including without limitation any implied warranty of merchantability or fitness for a particular purpose. In no event will IBM be liable to you for any damages, including any lost profits, lost savings or other incidental or consequential damage arising out of the use or inability to use any information provided through this publication even if IBM has been advised of the possibility of such damages, or for any claim by any other party.**

Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

This publication may contain technical inaccuracies or typographical errors. Also, illustrations contained here may show prototype equipment. Your system configuration may differ slightly.

This publication may contain articles by non-IBM authors. These articles represent the views of their authors. IBM does not endorse any non-IBM products that may be mentioned. Questions should be directed to the authors.

This information is not intended to be an assertion of future action. IBM expressly reserves the right to change or withdraw current products that may or may not have the same characteristics or codes listed in this publication. Should IBM modify its products in a way that may affect the information contained in this publication, IBM assumes no obligation whatever to inform any user of the modification.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such products, programming or services in your country.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

All specifications are subject to change without notice.

To correspond with the *IBM Personal Systems Developer*, please write to the Editor at IBM Corporation, Internal Zip 4607, P.O. Box 1328, Boca Raton, FL 33429-1328.

Trademarks

IBM, Personal Computer AT, Personal System/2, PS/2, OS/2, AIX, QuietWriter, Micro Channel, and DisplayWrite are registered trademarks of IBM Corporation.

PC XT, AS/400, Personal Computer XT, Systems Application Architecture, SAA, Operating System/2, C/2, and Proprinter are trademarks of IBM Corporation.

COMDEX is a trademark of the Interface Group, Inc.

Ethernet is a trademark of Xerox Corporation.

UNIX and Touch-tone are registered trademarks of American Telephone and Telegraph Corporation.

Microsoft, MS, MS-DOS, Xenix, and CodeView are registered trademarks of Microsoft Corporation.

Lotus and 1-2-3 are registered trademarks of Lotus Development Corporation.

Macintosh and Mac are registered trademarks of Apple Computer, Inc.

PageMaker, Aldus, and Aldus FreeHand are registered trademarks of Aldus Corporation.

Intel, i486, and i860 are trademarks of Intel Corporation.

WordPerfect is a registered trademark of WordPerfect Corporation.

PostScript is a registered trademark of Adobe Systems Incorporated.

VMS and Ultrix are registered trademarks of Digital Equipment Corporation.

Dataquest is a registered trademark of Dataquest Incorporated.

MKS Toolkit is a trademark of MKS Products, Inc.

CompuServe is a registered trademark of CompuServe Incorporated.

Turbo C and Borland International are registered trademarks of Borland International, Inc.

Rockwell International is a registered trademark of Rockwell International Corporation.

NCR is a registered trademark of NCR Corporation.

Unisys is a registered trademark of the Burroughs Corporation.

Editor's Comments

The reader survey cards from our Fall '89 issue told us you liked all of our departments, with some ranked higher than others. Your favorites are Software Tools, Presentation Manager, IBM Directions, SAA, LANs, and Developer Assistance Program News. In this issue we'll continue those themes with special emphasis on OS/2 2.0.

32-Bit OS/2 Version

We introduced a new IBM/Microsoft Software Development Kit (SDK) for the 32-bit version of OS/2 in the Winter '90 issue. Many of you are already using that SDK. While an SDK is certainly far from the final "street version" of OS/2 2.0, it does contain early system code, software tools, language compilers, and documentation, which can give you a head start on the application development/conversion process.

We asked our staff in the Boca Raton Programming Center to describe the important new features in the new 80386/80486-based OS/2 version. They contributed five excellent articles covering everything from an overview to specific function calls. This includes the flat memory model's 32-bit addressability, 80386 paging, the new application programming interface, compatibility with OS/2 1.X, multiple virtual DOS machines (virtual 8086 mode), virtual device drivers, numeric processor support, and programming tools. In future *Developer* issues we'll bring you more on this exciting new application platform.

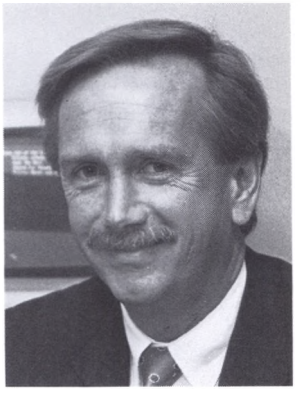
Spotlight: PageMaker for OS/2

Beginning with this issue, we will visit some leading-edge OS/2 software vendors. This provides a good chance to see how a well-known developer looks at OS/2 and its market, and how they managed the formidable task of converting a DOS application to OS/2.

We'll launch our Spotlight feature with Aldus® Corporation and their PageMaker™ for OS/2. This is a non-trivial application, posing special challenges in a conversion to any new operating platform. The Aldus team decided early to capitalize on the unique graphical and multithreading features of OS/2, which improved the usability, function, and performance of PageMaker in the process. Their approach involves a balanced mix of platform-specific "edge" code and portable "core" code. In addition to discussions with key members of the PageMaker development team, we talked to Aldus founder and president Paul Brainerd to get his perspective on application development in the 1990s.

We are also running our first articles on performance tuning and national language support. Everyone wants the best performance from their applications, but how many U.S. developers consider the worldwide software market? OS/2's National Language Support (NLS) can make programs more translatable and installable in other countries.

Dick Conklin, Editor



Dick Conklin

Blue icons indicate a technical article.

Yellow icons indicate a marketing article.

CORRECTIONS

Winter, 1990, page 72: Figure 1, Limit for Maximum Windows in System should read "1200 for OS/2 1.1, 700 for OS/2 1.2."

Winter, 1990, page 55: The BBS has moved and the telephone number has changed from (404) 988-2790 to (404) 835-6600.

Developer Assistance Program

Developer Assistance Program Update

Don Barovich

by Don Barovich

The IBM Developer Assistance Program provides a variety of support services for companies that are writing OS/2 applications for resale.

1. **Technical Support**: Support via IBMLink electronic interface to IBM support organization. It is available twenty-four hours each day, seven days a week.

2. **Defect Support**: A toll-free number now available to report defects and receive the latest corrective service diskettes.

3. **Software Discounts**: 40% discount on selected OS/2 products ordered directly from IBM.

4. **Hardware Rebates**: 5–10% rebates on selected PS/2 products purchased from IBM authorized dealers.

5. **IBM Personal Systems Developer:** Free subscription to this quarterly technical publication.

6. **Code Downloading Service:** Access to sample programs, demonstration packages, and other productivity aids. Some sample programs and listings contained in the *Developer* will be available through this service.

7. **Technical Education**: Invitations to participate in IBM-sponsored technical seminars and workshops.

8. **Marketing Support**: OS/2 application marketing support through listings in the *OS/2 Application Guide.*

If you are not a member of the IBM Developer Assistance Program, you may request an enrollment form (described on the next page).

ESDTOOLS GOES ONLINE

The new ESDTOOLS service is now online, providing companies enrolled in the Developer Assistance Program with access to OS/2 productivity aids. If you have an OS/2 software tool to contribute or would like to access this system, call (407) 982-6408 for more information.

OS/2 APPLICATION GUIDE

A new version of the *OS/2 Application Guide* is now available. This version of the *Guide* differs from prior versions in that it contains a description of currently available applications. The new *Guide* contains descriptions of over 600 available OS/2 applications.

An online version of the *Guide* is also provided. You may access the online version from any system with a standard asynchronous modem (300 to 9600 baud) by calling (404) 835-6600 with communications software set to 8-N-1. No registration is required.

To include your OS/2 application in the online database and in future issues of the *Guide*, please call us at (407) 982-6408 to request an application form.

PM CONVERSION WORKSHOP

Our DOS-to-OS/2 workshops continue to be popular with Developer Assistance Program members. Software developers from all over the country have taken advantage of this unique opportunity to "jump-start" their OS/2 programming projects. They have brought their commercial PC/DOS applications to our Developer Assistance Lab in West Palm Beach, Florida, and with the help of a knowledgeable and helpful staff, have ported them to OS/2. This highly productive "hands on" concept has opened the door for OS/2 programming to more than 100 software developers (see *IBM Personal Systems Developer*, Winter 1990). While the workshops have shown developers the advantages of programming in an OS/2 environment, they have also generated interest in developing a PM interface for their new OS/2 applications.

In response to that interest, a Presentation Manager (PM) workshop was successfully piloted in December, using a lecture-lab format similar to the one for the original workshop. To qualify for an invitation to the PM Workshop, developers must have a commercial application that operates in an OS/2 full-screen or text-mode window. Attendees bring their products to the workshop and, during a combination of lecture and lab sessions, develop a PM interface for them. Currently, two or three workshops are taught each month, with an equal number of DOS-to-OS/2 and PM sessions planned for this year.

If you have any questions about the DOS-to-OS/2 or Presentation Manager workshops, please call the Developer Assistance Program at (407) 982-6408, and someone from the workshop staff will get in touch with you.

ELIGIBILITY AND ENROLLMENT

The IBM Developer Assistance Program is for software developers working on products for commercial release. You are eligible to participate in the program if:

- You develop products that support IBM OS/2 or IBM DOS.
- Your company is a U.S. company or the U.S. subsidiary of a non-U.S. company.
- You are currently marketing your products.

If you are not currently marketing a product, you may submit a non-confidential, detailed business plan showing marketing and development activities and schedules. IBM reserves the right to accept or reject an application based on this business plan. There is no charge for enrollment in the Developer Assistance Program.

To request an application form or ask questions about the program, contact Developer Support:

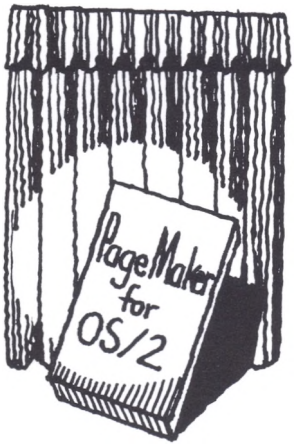
IBM Corporation
Developer Support
Internal Zip 4605
P.O. Box 1328
Boca Raton, FL 33429-1328
(407) 982-6408

Don Barovich, *IBM Entry Systems Division, 1000 NW 51st Street, Boca Raton, FL 33429.*
Mr. Barovich is Manager of the Developer Assistance Program and Developer Support Strategy. He has been with Entry Systems Division since it was formed and has worked on PC software since 1982. He has a BS in Electrical Engineering from Penn State University.

The IBM Developer Assistance Program can be reached at (407) 982-6408.

Spotlight

PageMaker 3.0 for OS/2



In this issue of the Developer, we launch our new Spotlight feature, in which we look at a leading-edge OS/2 application and the people who wrote it. Our first Spotlight product is PageMaker for OS/2. We visited Aldus to find out what they think of OS/2, how they tackled the development of one of the first OS/2 applications, and what they see for the future. We thank everyone at Aldus who helped us put this together, and we hope our readers find it interesting and informative.

PageMaker 3.0 for OS/2 is one of the first major applications designed specifically for the OS/2 environment. Although PageMaker 3.0 for OS/2 looks almost exactly like the Windows version, Aldus has made significant internal changes to take advantage of OS/2. PageMaker is very demanding of the operating system; what appears on screen must appear on paper from the printer (screen and printer fonts are mapped). The product has been selling well since first shipping on September 20, 1989.

THE OS/2 VERSION OF PAGEMAKER

The engineering team was given the charter to do what was necessary to take advantage of OS/2 in moving PageMaker for Windows onto the OS/2 platform. The value added by

The engineering team was given the charter to take advantage of OS/2.

PageMaker 3.0 for OS/2 Engineering Team

Product	
Name:	PageMaker 3.0 for OS/2
Category:	Desktop Publishing
Price:	$795
First Ship:	September 20, 1989
Suggested: Memory:	4–6 MB, including OS/2
Other Platforms:	DOS/Windows Macintosh
International:	Sold in over 40 countries

PageMaker 3.0 for OS/2 shows the benefits of OS/2 very well. Team members identified the strengths of the operating system and then decided what functions would best exploit them. They concluded that multitasking was the key strength for PageMaker.

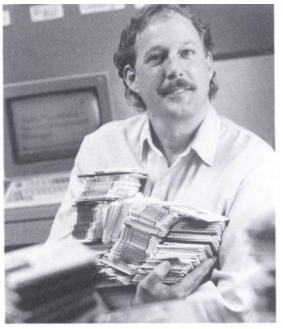
Multitasking

The biggest benefit PageMaker for OS/2 offers users is the speed of interaction. Preemptive multitasking in OS/2 allows smooth, responsive interaction with the user (during text entry, for example) while significant tasks are being performed in the background. OS/2 effectively eliminates the hesitation seen in DOS or Windows applications that perform activities in the background.

According to Peter Kron, principal software engineer and one of the original designers of PageMaker 3.0 for OS/2, preemptive multitasking efficiently provides applications that are responsive to users—where users can interrupt a long activity if they decide to do something else. While DOS or Windows applications could theoretically poll for keyboard input, the overhead for that would be prohibitive. "I'd like the system to do system functions and let the application focus on adding value. If I am going to spend two weeks coding something, I'd like it to be where I am creating some uniqueness, not building services that the system could provide," said Mr. Kron.

Peter Kron

Besides the time required to perform a task, another measure of speed is how much of the time the hourglass is displayed. Multitasking can hide long processes in the background and make the program more available to the user. The developers of PageMaker 3.0 for OS/2 realized that minimization of the time the hourglass is displayed is an important objective. Mike Sherwood, product manager of PageMaker 3.0 for OS/2, explains why a new icon was added: "The Busy icon appears when the cursor is positioned over a process that is busy, thus encouraging the user to start an alternative process. We wanted to allow the user to *drive* the application, not *wait* for the application."



Mike Sherwood

The biggest benefit PageMaker for OS/2 offers users is the speed of interaction.

Peter Kron has another view of the value of multitasking. "The advantage is not in using more than one application at the same time. That sounds hard, even to me. The advantage is in eliminating the hourglass. Activities like printing or importing take some time, but with OS/2 the user can work on something else during a long operation. As another example, suppose you are in the middle of an application and someone comes into your office and asks for something that requires you to run another application. That is easy to do with OS/2 without completely stopping what you are doing."

PageMaker 3.0 for OS/2 uses multitasking in these five areas:

- **Initialization of the program**: The first screen and menus come up while code is being loaded in the background. The user can select commands, such as loading a file. By the time the file is loaded, the rest of the program is loaded too. This makes the program feel much snappier.

PageMaker uses multitasking in five areas.

- **Loading text and graphics**: Text is loaded to be flowed onto the page in the background. In the Windows version, the user must wait for the file to be fully read in. In OS/2, the user can start to work with the document as soon as the first page is available.
- **Autoflow of text**: PageMaker will create new pages and flow text into defined margins automatically while the user continues to work in another publication. Under Windows, the user must wait until all text has been placed before beginning another task.
- **Printing**: One publication can be sent to the printer while the user clicks on another publication or application and continues to work while the publication prints. This is a time saver for long or graphics-intensive publications.
- **Screen repaint**: A user can interrupt screen repainting and immediately begin another task.

User Benefits of PageMaker 3.0 for OS/2

The techniques employed in PageMaker 3.0 for OS/2 provide the following benefits to users:

- **Performance enhancements**: Launching a publication is 42% faster than in the Windows version. Saving a publication to disk is 40% faster. Editing text is 20% faster. One user of PageMaker 3.0 for OS/2 reported that he produced a calendar in a third of the time by taking advantage of the multitasking capabilities. Figure 1 shows a comparison study of the speed of the OS/2 and Windows versions of PageMaker.
- **Improved imports**: Metafiles no longer have a 64 KB limit; therefore, large CAD drawings can be imported. Mainframe graphics (PIF/GDDM) are supported via OS/2 1.2.
- **Multiple publications**: Up to four publications can be opened from the same instance of PageMaker. This makes view-

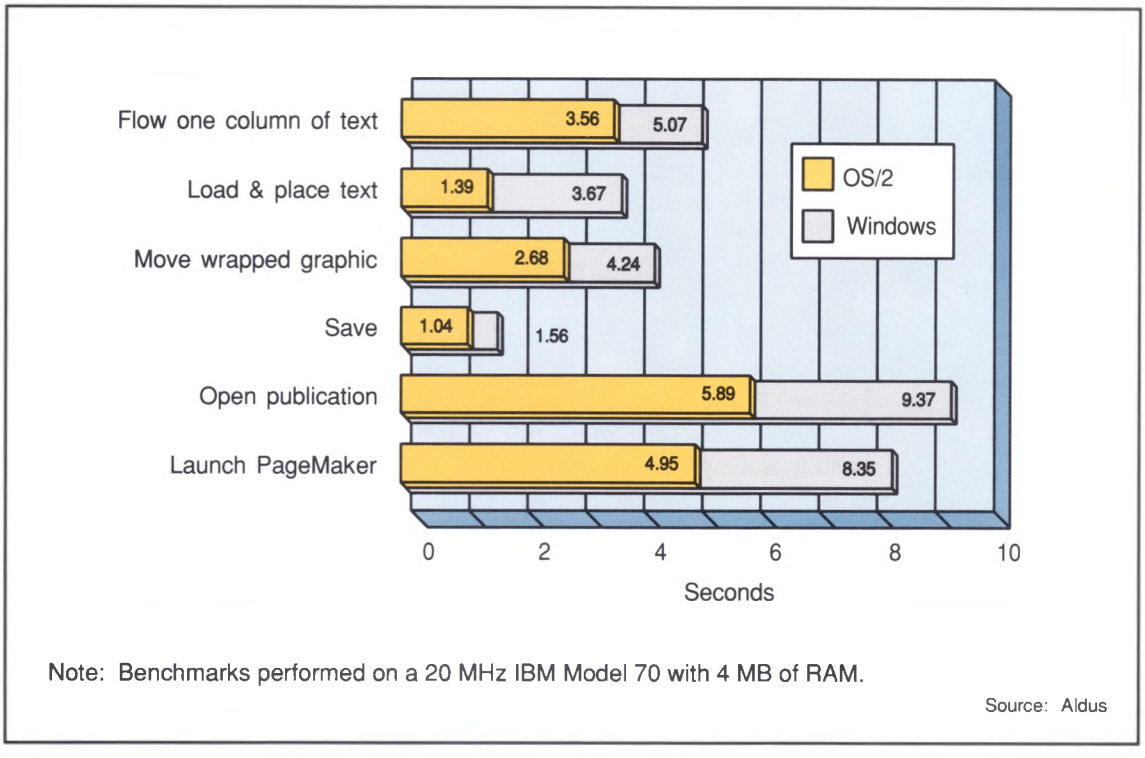


Figure 1. Comparison of Speeds: Windows and OS/2 Versions of PageMaker

ing multiple publications or cutting and pasting between publications easier.

- **Smart clipboard**: An enhanced interface to the clipboard keeps text/graphics formatting information intact for PageMaker publications as well as the Rich Text Format (RTF). RTF support allows charts cut from Microsoft Excel to come through with text formatting intact.

- **File compatibility**: PageMaker 3.0 files from the Macintosh and Windows versions can be opened by PageMaker 3.0 for OS/2 and vice versa. Text, formatting, layout features and most graphics remain intact.

- **OS/2 1.2 support**: A planned second release (3.01) of PageMaker for OS/2 will support CGM and GDDM/PIF graphic file imports, hierarchical menus, the High Performance File System, and long file names. The update will be provided free of charge to all registered owners of PageMaker 3.0 for OS/2.

THE CONVERSION TASK

Changing Windows PageMaker, which is single-threaded, into an application consisting of five threads was challenging. It was also a significant amount of work. As Mike Sherwood recalls, "It was certainly exciting on the engineering side to make it work. In the process of doing that, anyone who used PageMaker for OS/2 could see the benefits."

How It All Began

In November 1987, IBM and Microsoft approached Aldus to help them get the new operating system "off the ground." They were going to use PageMaker for early systems testing of OS/2. Aldus showed a two-thirds functioning version of PageMaker 1.0 under OS/2 at COMDEX/Spring in 1988. This effort was based on PageMaker 1.0, while PageMaker 3.0 was being developed in parallel. In July 1988, after the Windows PageMaker 3.0 code was available, four programmers began working on PageMaker for OS/2. They concentrated on where they could add value: half of the team focused on threads and multitasking; the others worked on basic porting tasks, such as GDI (the Windows screen mechanism) to GPI (the OS/2 screen mechanism).

We asked Mike Sherwood why they decided to embark on an OS/2 development effort so early. He said, "Part of the reason was to get engineering up to speed on OS/2. Another reason was to find out from users how they expect to use OS/2, and determine their environment. We validated that OS/2 applications deliver significant features over DOS and Windows. In the end, we did it because the user gets many benefits from PageMaker for OS/2. It's one of the best examples of a multitasking application available today."

PageMaker was originally developed for the Macintosh®. When Aldus decided to support Windows, PageMaker core code was rewritten to be portable between the Mac® and Windows. That was a big advantage for the OS/2 team, because all the Windows code had already been isolated. Two areas that were changed for OS/2 were window management and GPI graphics Application Program Interfaces (APIs). According to Peter Kron, "The OS/2 PM APIs are very similar to the Windows API; most of the difference could be handled by compiler macros. Imaging, however, was a different story." The imaging in PageMaker already had been modularized down to a fairly low level because there were even bigger imaging differences between the Mac and Windows. In some cases the developers just put in compiler conditional code to handle OS/2 and Windows. "Sometimes the Windows code was imbedded in other code, which made us realize that the base wasn't as portable as we thought. So we replaced these sections with a more modular design."

PageMaker for OS/2 consists of five threads.

According to Kron, multiple PageMaker processes can run concurrently and communicate through shared memory. "A Windows menu identifies which publications are active in the other processes. We did that because it separated the memory usage, making it much easier to support multiple activities in the applications. The user sets up one process, for example, to autoflow a document, and switches to another process to do editing. There is no interference between the two."

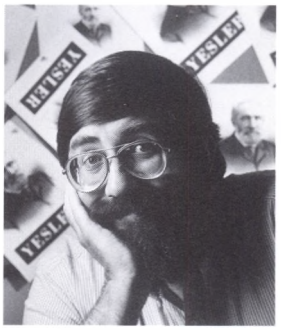
"With multithreading," Kron continued, "the developer must worry about many concurrent activities. We limited multithreading to areas that currently take a long time, such as printing and text flow. Under Windows, long operations prevented the program from getting back to the message queue quickly enough. Multithreading gets around that and allows the user to perform another task. We hope to implement many other ideas for multitasking in future versions of PageMaker for OS/2."

ORGANIZATION OF PAGEMAKER CODE

Aldus has developed code on a "core and edge" model.

From the beginning, Aldus has developed code on a "core and edge" model. Core code is portable across platforms. Edge code is platform-specific. The problem with this approach is that as operating systems evolve on each platform, less "least common denominator" function is left. Because coding to a least common denominator would expose the company to competition focused on one particular environment, Aldus codes to exploit the capabilities of each platform to the greatest extent. The impact tends to make the edge code larger. Today, edge code represents about 35–40% of PageMaker 3.0 for OS/2; much of it relates to performance. Dick Mathews, vice president of engineering, said, "The edge percentage will decline slightly over time as the environments come to support similar capabilities."

The engineering group is not organized around core and edge code, but rather by function. Most engineers on the application teams understand the two older environments (Macintosh, DOS/Windows), and knowledge about the OS/2 environment is now spreading through the organization. "We don't want to hide the capabilities of any platform from the application programmers," says Howard Campbell, engineering lead. The application teams try to develop core code whenever possible.



Howard Campbell

Aldus emphasizes making applications the best they can be on any given platform. But they do develop virtual components, such as printing, imaging, font handling and resource access, that exist within products in different platforms. As shown in Figure 2, these hook into the larger core code of each product that does the data manipulation.

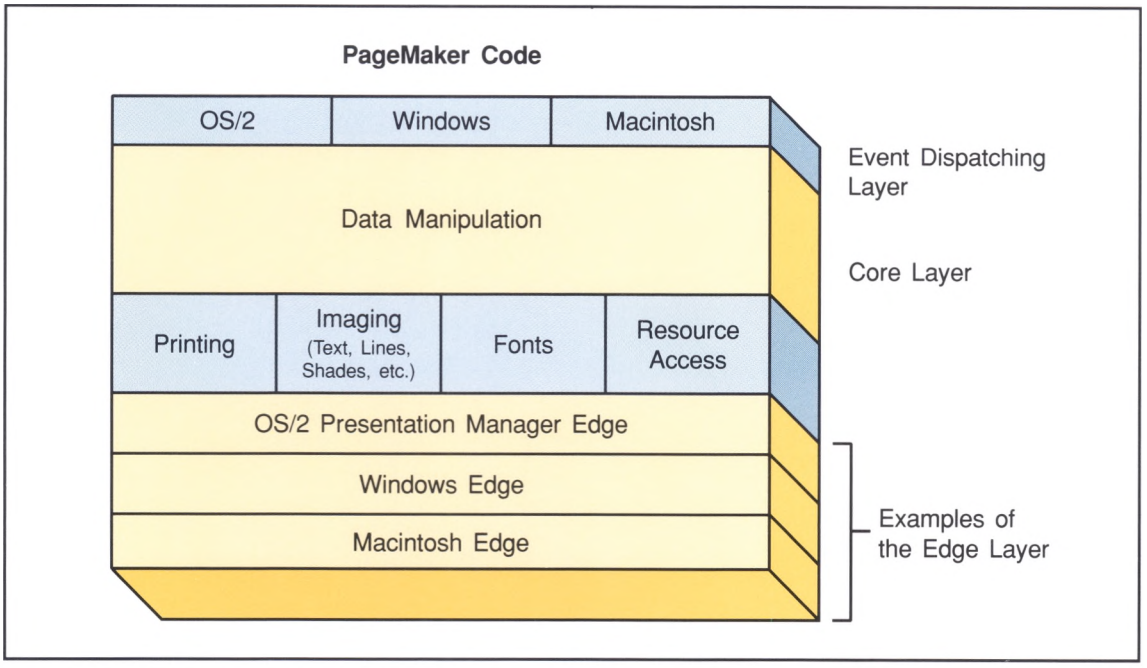


Figure 2. Organization of PageMaker Code

There is an engineering team that provides common components that span products and platforms. As shown in Figure 3, these components are sometimes installable, such as import and export "filters."

Programmers are not dedicated to either core or edge code alone. For example, a text composition programmer writes some core and some edge code for Windows and the Mac. The fundamental requirements for text composition do not change across platforms. "The platform experts tend to be used as consultants rather than for writing code. As we develop more of these virtual components in the future, more will be available to the application writers," Campbell said.

We asked Campbell about the impact of threads on the core code concept. He felt it simply added an additional design constraint. "The functionality being provided by the core was extended. The single biggest problem in converting PageMaker to support OS/2 was synchronizing access to data structures. In the single-threaded world, any code that accesses data has exclusive access. Under OS/2, that is not true." The problems encountered and the added development time result from: 1) understanding where the points of synchronization have to be, and 2) finding ways to coordinate access to them.

Other than creating these new coordination mechanisms, Aldus programmers did not change the core code significantly, but did restructure it.

"We now understand better the issues about multitasking. Future applications will be designed with those constraints in mind—more independence between functional components than in the past," said Campbell.

Campbell also alluded to the future of PageMaker for OS/2: "Connectivity to mainframes is an area for enhancement. OS/2 is a very strong networking environment. Our customers would like PageMaker to be able to interface with mainframes to pull down large GDDM graphic files."

Use of OS/2 for Development

Development groups tend to move to new computing environments earlier than the mainstream. Internally, Aldus has switched to the OS/2 environment for Windows PageMaker development, primarily because of the increased memory and the performance achieved by multitasking. For example, a programmer can build PageMaker while continuing to work on other files. Synchronization with the central code repository can be done in the background. "I think that more and more developers will reach the same kind of limitations that we did, and move to OS/2," said Campbell.

Aldus has switched to the OS/2 environment for Windows PageMaker development.

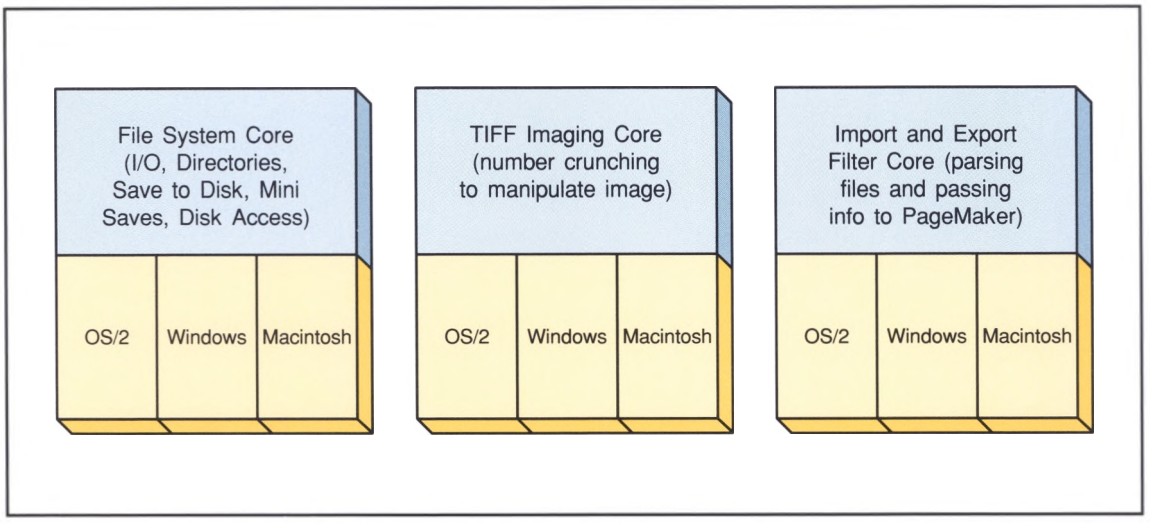


Figure 3. Examples of Core Components

CodeView®, the primary tool used, made debugging the OS/2 version of PageMaker much easier than it was under Windows. "During the beta cycle, we did not have any problem that lasted more than three or four days. We could not say that about the previous Windows releases," Campbell said.

TIPS FOR OTHER DEVELOPERS

We asked the Aldus developers and management what advice they would offer to a small programming group just getting started with OS/2.

"Don't just do a port," cautioned Campbell. "Take the training from IBM and Microsoft. Use online support; it tends to be more current than the written documentation. It took us awhile to understand the design implications of multitasking. Those need to be clearly understood before you start, particularly if you are embarking on a new application. The application must be partitioned into blocks that can be performed concurrently. That may seem obvious, but because of the bias we all have from our experience programming in single-threaded environments, it is not. Most of us see computers as serial processor machines. It's hard to break up the application and recognize where you are going to communicate. You have to design how tasks will communicate with each other from the beginning; the flowchart is not the place to start. The problems with data structure access can be a surprise if, all of a sudden, you realize that another person is not doing it properly. If you approach this from the traditional computing background, you will end up with the same porting problems that we did."

"Don't just do a port," cautioned Campbell.

Mike Sherwood said that, although the big software companies will all have OS/2 versions of their products, "there are opportunities for small companies to develop connectivity utilities to enhance productivity in that environment. Take advantage of the operating system. Look for ways to add value to the operating system, for example, by providing data translation utilities and bringing in text and graphics from larger systems."

Peter Kron advised OS/2 programmers to "immediately think multithreading. If you think you will ever get into multithreading, learn what it is, and learn how it works in OS/2. It is not expensive to provide for multithreading if you do so from the start, but the synchronization between data structures in multithreading gets very complicated and may be impossible if you try to add multithreading later. We were limited to areas where structures were not receiving concurrent access. There were other things I would have liked to do, but the structures were not set up to do multithreading.

"There is some cost to this: there is some extra code to synchronize threads, and debugging is more difficult because sequence-dependent bugs may not be repeatable. By putting everything into global data and not thinking about it until later, you will probably face a rewrite someday."

MARKETING

Mike Sherwood has been the product manager for PageMaker for the past four years. He is now product manager for PageMaker 3.0 for OS/2. We asked him to tell us about the market segments that the OS/2 version addresses, and the product's success to date.

"At first, the OS/2 version of PageMaker will appeal to dedicated desktop publishing users. They can see the benefit of performance, and because their jobs are dedicated to PageMaker, they can justify the added cost of the system. That will change over time. A casual user market for PageMaker for OS/2 will exist when a critical mass of OS/2 applications (word processing, spreadsheet, EMail) is shipping, and OS/2 is being ordered for a large number of new machines. Dedicated users are not found only in large installations. We even have a wine-and-cheese shop in Bellevue (where else?) that uses it because of the productivity gains."

Sherwood continued, "Europe is a huge market for OS/2 and PageMaker. In Europe, the power publishing platform consists of PCs, many of which are connected to mainframes. In the U.S., there is more diversity, including various brands of PCs and workstations. But IBM has been very effective at marketing solutions in Europe."

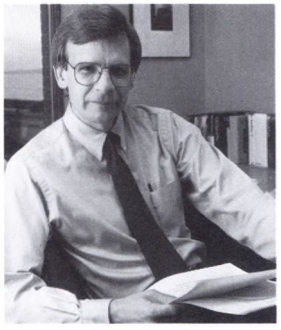
When asked if sales have met expectations, Mike said, "Yes, we have met our goal in the U.S. and surpassed it in Europe. We expected healthy sales in Europe because of the positive response that PageMaker 3.0 for OS/2 received during the test phase. In addition, Europe accounts for half of Aldus' overall sales. But actual results have bettered even these expectations."

Typically, Aldus Europe does the language translations, and offers versions in most European languages. This process is fairly straightforward because Aldus has an architecture that supports localization. Applications are built in the U.S. so that they can be localized by developers in the country. This version of PageMaker for OS/2 was written to support character sets for most languages that have code page support in OS/2. PageMaker 3.0 for OS/2 uses code page 1004 for its native activities.

Aldus is continuing to collect information from PageMaker for OS/2 users. In the back of the "Evaluating OS/2" booklet, Aldus asks users to describe their environment, why they bought the product (speed, networking), and so on. The company is following up by phone to determine why and how users are using PageMaker; the information will help Aldus design the next version. The next version will be based on what their users need, combined with the added capabilities of OS/2 2.0.

MANAGEMENT PERSPECTIVE

The fact that PageMaker 3.0 for OS/2 is one of the first PM applications is not surprising. PageMaker was also one of the first major applications to run under Microsoft Windows. Paul Brainerd, founder and president of Aldus, encouraged the development of PageMaker 3.0 for OS/2 to take advantage of the OS/2 environment, not be just a port from Windows.



Paul Brainerd

Developer: Could you give us some of your philosophy behind PageMaker?

Brainerd: The basic idea behind PageMaker is simple. We want to integrate text and graphics. From the beginning, we needed a graphical environment to accomplish this. In early 1984, we built a prototype of PageMaker on the Macintosh. That product was much better than we had anticipated. The Mac product helped us to become believers in the standard graphical environment, which provides wide acceptance and ease of use. Many DOS developers are not yet sold on the value of a standard graphical interface. We (and other large software companies) believe that we gain a great deal by conforming to a standard user interface, rather than developing our own.

We want to integrate text and graphics.

Developer: Some developers think that they lose their differentiation by conforming to a standard user interface. How would you answer that?

Brainerd: Maybe so, but remember our expert knowledge of the application adds to our value. That's what users are paying us for. It makes us work a little harder to deliver value. Even though the Macintosh has a standard user interface, we were innovative in our use of that interface. We were the first product to have a floating toolbox on the screen that could be positioned anywhere by the user. We took the horizontal scroll bar that normally goes across the entire bottom of the screen and shortened it to allow room for direct manipulation to go to any page. A lot of information is conveyed there. Those were extensions to the standard that we think are consistent, yet innovative.

Developer: What are your thoughts about OS/2?

Brainerd: When OS/2 Presentation Manager was announced on April 2, 1987, it was what we were looking for on the PC. Even though acceptance has been slower than we expected, we believe that it will become the standard graphical environment on the PC. The Windows version of PageMaker has been limited by the 640 KB limit of DOS. It is a very hard environment in which to program a large application like PageMaker. Since OS/2 provides a standard graphical environment

and access to vast amounts of memory, we committed very early to OS/2 development.

Developer: Do you see any new distribution issues for the OS/2 version of PageMaker?

Brainerd: We are working closely with resellers (and the IBM direct sales force in Europe) to make PageMaker 3.0 for OS/2 part of the initial sale of OS/2. We get very good attention from the IBM marketing organizations because PageMaker helps sell OS/2 and IBM hardware. In many cases, we are doing joint sales calls. This is the advantage of being on OS/2 first.

"Seek out applications that could be done only on OS/2 2.0."

This works particularly well in Europe because the system solution sales approach is used more there than in the U.S. Sales people in Europe demonstrate applications running together more than in the U.S. As a result, European customers have a much greater understanding of OS/2's benefits, such as multitasking. In much of the U.S., multitasking is still a theoretical benefit; in Europe, however, potential buyers understand multitasking's potential impact on their productivity because it has been demonstrated to them.

Developer: Do you offer special pricing for Windows PageMaker owners to purchase the OS/2 version?

Brainerd: Through February 1990 (the first six months of the product), we allowed users to obtain PageMaker 3.0 for OS/2 as an additional product for $95. I say "additional" because they can keep their DOS/Windows product.

Developer: What advice do you have for others looking at the OS/2 opportunity?

Brainerd: If you are starting today with a significant application that will come to market within a two-year timeframe, then I urge you to look at OS/2 2.0 (32-bit). If a product will be out sooner than that, you need to look at 16-bit OS/2 and Windows. If you are a large software developer, you might be able to put in a layer to share much of the source code between Windows and OS/2. If you are small or a start-up, go with 32-bit OS/2, get it out as soon as you can, and ride the wave of OS/2 enthusiasm.

Look at applications that people associate with workstations and UNIX. OS/2 may replace UNIX as the multitasking workstation platform. Think about a fast 386 or 486 PC running OS/2 doing the same thing for much less cost.

If you can get to market next week, you probably want to be on Windows rather than OS/2. What about the 6–12 month timeframe? How clear is your crystal ball? If your application is not constrained by the limitations of DOS, you may want to go with Windows. On the other hand, you should be on a platform that has as much "headroom" as possible.

Developer: Could you compare the various graphical environments for us?

Brainerd: The GPI environment of the Presentation Manager is much richer than the Windows GPI. It has the GDDM mainframe standard driving it, and allows for many more real-time transformations (needed for CAD applications). GDI is similar to Macintosh-level graphics. You must work really hard on the Macintosh to produce something sophisticated like our Freehand™ product. On GPI, a CAD package would be a natural.

Developer: Where are the opportunities for new products?

Brainerd: Seek out applications that could be done only on OS/2 2.0—those that cannot easily be done on the Mac or Windows. That might be CAD (taking advantage of graphics capability), mainframe database interfaces (taking advantage of connectivity and background processing), document storage/retrieval (background scanning and indexing), EMail (host connect and peer-to-peer) and, of course, publishing.

Within publishing, connections to databases are still not as good as we would like. We have some import filters, but that's just the beginning. We could use a Structured Query Language (SQL) interface to select mainframe text or graphics to download into PageMaker. Many other mainframe connectivity and file-format conversion opportunities will probably arise as more micros are connected to mainframes. Network administration within publishing or documentation environ-

ments is an opportunity; for example, to make sure that fonts get propagated properly. It is difficult to get all machines configured properly so that documents can move among many machines in a group. Intelligent print utilities that give feedback to the user or operator when they need letterhead loaded, for example, are still needed.

Developer: What advice would you offer to a small programming group just getting started?

Paul Brainerd: Embrace the user interface standards, and believe that they will help your customers (reduced learning curve, training, and ongoing support). Support the standard data interchange formats like metafiles and the clipboard Rich Text Format (RTF), and import and export standard formats.

OPPORTUNITIES FOR OTHER DEVELOPERS

PageMaker is an integrating application. Other software developers interested in integrating their applications with PageMaker 3.0 for OS/2 should contact David Cloutier, Manager of Third-Party Relations. Aldus has defined a data interface and protocol that can be used by other programs wanting to exchange text and graphics with PageMaker.

Opportunities exist for third parties to provide filter and import utilities. Synergism exists with applications that create text or graphics, such as draw programs, databases, and spreadsheets. In the future, Aldus will be doing more with color. Connectivity to mainframes (formats such as GDDM, and applications like DB2) and other environments is also needed.

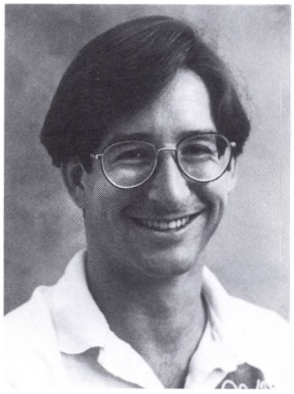
Aldus Corporation

Founded:	1984
Employees:	500+
Distribution:	8,000 dealers in the U.S. 3 international subsidiaries
First PC product shipment:	PageMaker for MS Windows, January 1987
Other products:	Aldus FreeHand™ - a drawing program for the Apple Macintosh
	Aldus Persuasion™ - a desktop presentation program for the Macintosh
	Aldus PageMaker 4.0 for the Macintosh (available 2nd quarter 1990)

"Embrace the user interface standards, and believe that they will help your customers."

32-Bit OS/2

OS/2 2.0 Overview



Michael S. Kogan

by Michael S. Kogan

OS/2 2.0 is the first version of the OS/2 operating system to exploit the 32-bit features of Intel 80386 systems. It is designed to run on both 80386 DX and SX systems, as well as 80486 platforms. The system uses the paging features of the 80386 to provide a demand paged, virtual storage environment that supports a new 32-bit programming model. The 32-bit programming model enables applications, subsystems, and the system itself to utilize the 32-bit register set, and 32-bit instructions and addressing modes, as well as memory objects larger than 64 KB. OS/2 2.0 provides binary compatibility for OS/2 1.X applications and subsystems. The DOS-compatibility environment has been enhanced to take advantage of the virtual 8086 mode of the 80386, enabling multiple DOS sessions with background execution. The system also provides a system-wide 80387 numeric coprocessor emulator and support for both the High Performance File System (HPFS) and the File Allocation Table (FAT) file system.

32-BIT PROGRAMMING ENVIRONMENT

The overriding goal of the 32-bit programming environment is to provide an architecture that allows applications, subsystems, and the system itself to be portable to processing platforms other than the single processor Intel 80X86 machines. This requirement led to the development of a new memory model called the *flat model*, which enables processes to view memory as a large linear address space addressable by 32-bit offsets. The flat model architecture is easily portable to most processing platforms since the hardware must provide only a base register capable of addressing a large-paged linear address space and an offset register for indexing into the address space. OS/2 2.0 was not designed to be 386-specific, but rather 32-bit OS/2 implemented on the 80386 platform.

Flat Memory Model

The flat memory model, as implemented on the 80386, is also known as the "0:32 model" because only the 32-bit offset into the process address space is used to develop the address of a single byte of memory. OS/2 1.X uses the 16:16 model for addressing memory, in which a segment and the offset into that segment must be specified in order to address a single byte of memory. The flat model effectively hides all segmentation from the 32-bit programmer, resulting in a portable programming model with much higher performance than a segmented system could provide.

In OS/2 1.X, a process address space consists of a collection of segments mapped by a Local Descriptor Table (LDT); a byte addressable segment is the basic unit of allocation and sharing. The OS/2 2.0 process address space consists of a single, large ring 3 segment that the process addresses using 32-bit offsets. In the flat model, therefore, the basic unit of allocation and sharing is a 4 KB page. Instead of being divided into segments as it is in OS/2 1.X, memory is divided into memory objects that consist of one or more 4 KB pages. Unlike segments, memory objects are not relocatable, are allocated in units of 4 KB, and are aligned on page boundaries in the process address space.

The size of the process address space in OS/2 2.0 is 512 MB. This will be increased to the 4 GB limit that the 80386 supports in future OS/2 releases. This short-term restriction, discussed later in this article, is due to the OS/2 1.X compatibility. Figure 1 illustrates the process address space in the flat model.

Notice in the figure that the system is at the top of each process address space. Anytime an application attempts to use an address larger than 512 MB or an address that has not been allocated, an access fault is generated. The process address space itself is divided into private and shared regions, much like the LDTs in the OS/2 1.X system. Shared memory allocations grow down from the top of the address space whereas private allocations grow up from the bottom of the address space. The system guarantees that 64 MB each of shared and private memory are available at process load time.

Memory protection is one of the most significant differences between the flat model and the 16-bit segmented model. In the 16-bit segmented model, protection exists on a per-segment basis; access and limit checks occur on each segment selector load. Within the flat model, however, since all of an application's memory objects exist within the same segment, the Intel segment protection semantics are bypassed and page-level protection is used to manage the memory in the process address space.

When allocating memory using the flat model, an application can control the attributes of the pages within the range of addresses spanned by the allocated memory object. The commitment of memory to an address range may also be controlled by the application so that specific address ranges within a memory object can be made valid or invalid as the application sees fit. If an invalid page is addressed, an access fault occurs. Commitment, decommitment, and changing of the attributes of a page are done on a page granularity. To assist applications and libraries in their management of committed memory, the system exports the system page size (4 KB on an 80386). The following summarizes the attributes that can be associated with a single page or a group of pages within a memory object:

- **COMMIT**: Page is committed with backing storage.
- **DECOMMIT**: Page is not committed and access will cause an access fault.
- **EXECUTE**: Execute access to the committed page is allowed.
- **READ**: Read access to the committed page is allowed.

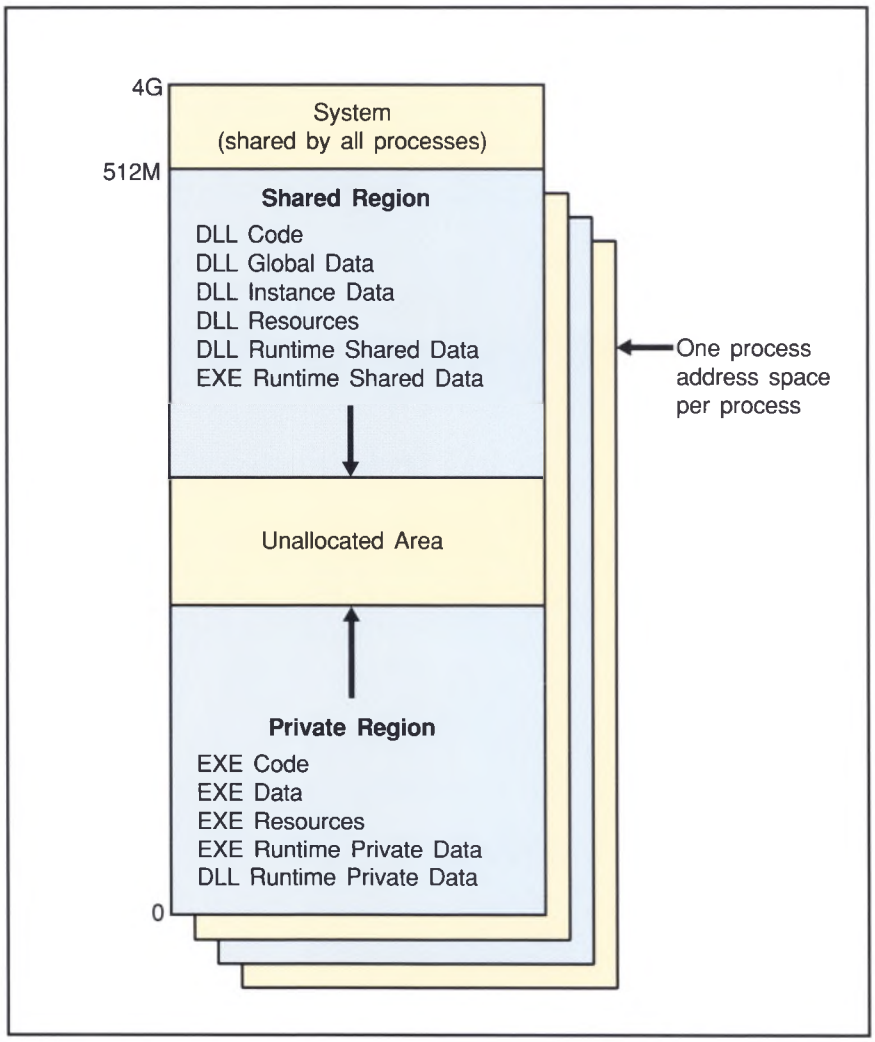


Figure 1. Process Address Space

- **WRITE**: Write access to the committed page is allowed.
- **GUARD**: The committed page is a guard page.

On the 80386, execute and read access are equivalent, and write access implies both read and execute access. The guard page attribute is used to facilitate automatic stack growth and stack limit checking, but may be used by an application in other data structures where appropriate. Since a memory object can contain some committed and some decommitted pages, "sparse" memory objects can be used with the flat model.

There is only one memory model for applications (small with no 64 KB restriction).

The high performance of applications, subsystems, and the system using the flat model is derived from several areas. In the segmented or 16-bit model, segment registers had to be reloaded with selectors every time a different 64 KB of memory needed to be accessed. These selector load operations are very expensive in protect mode on 80X86 processors because of the checking that must occur to ensure segment protection. In the flat model, a 32-bit offset relative to the base of the linear address space is used to address any byte of memory without reloading any selectors. In fact, 32-bit programs and subsystems do not use or know about the segment registers.

Another benefit of the flat model is that there is only one memory model for applications (small with no 64 KB restriction) instead of the many models (small, medium, large, huge) used in OS/2 1.X and all other systems that target 8088 and 80286 processors. The 32-bit registers and the arithmetic provided by the 32-bit Intel architecture also increase application performance.

Paging

While the paging feature of the 80386 is used to support the flat model and multiple DOS address spaces, it also allows OS/2 to provide a different kind of memory overcommitment than the 1.X system provided. In OS/2 1.X, segment swapping was used to keep the system running in memory-stressed conditions. Due to the I/O performance of most fixed disks, however, segment swapping does not perform well enough to provide generic virtual storage on demand. The 80386, in comparison, provides paging. From a system perspective, a lot of storage can be virtualized on fixed disk media at a much lower I/O cost because the size of a page is smaller than that of a segment. The system also can do a better job of tracking memory usage since memory aging algorithms operate on a page granularity instead of a segment granularity. For these reasons, OS/2 2.0, a demand paged virtual storage system, is designed so the system will run acceptably in nominally overcommitted situations.

Dynamic Linking

OS/2 2.0 provides shared library support in the form of 32-bit Dynamic Link Libraries (DLLs). The elimination of segmentation in the flat model is also reflected in the dynamic linking model of the system. Rather than all CALLs being "far," all code objects are "near;" segment reloading is not necessary when an Application Programming Interface (API) or dynamically linked routine is called. Therefore, the cost of making dynalink and API calls in OS/2 2.0 is significantly less than the cost of making comparable calls in OS/2 1.X. As in OS/2 1.X, DLLs execute in the context of the requester's thread.

The 32-bit DLLs are mapped into the shared region of the requesting process address space at load time and execute at ring 3 without Input Output Privileged Level (IOPL). This model's protection characteristics correspond closest to the ring 3 dynamic linking model in OS/2 1.X. Figure 2 illustrates how 32-bit DLLs are implemented in the flat model environment.

32-Bit API

OS/2 2.0 provides the 32-bit Standard Edition (SE) APIs to allow flat model applications to use the OS/2 system services. The 32-bit API has been designed so applications and subsystems that use and provide 32-bit APIs are source-portable to any future OS/2 2.X platform.

The API architecture in OS/2 1.X and OS/2 2.0 differs significantly. In OS/2 2.0, there are no more 64 KB restrictions, and API pointers are of the 0:32 format. The API calling conventions are different in order to support the

"near call" flat model, and the basic word size exploited by the API is the 4 byte dword. Many of the API names have been changed to provide more consistency, and several areas of the API have been enhanced to provide better functionality and to reduce redundancy. See the companion article, "OS/2 2.0 32-Bit API," in this issue for more specifics.

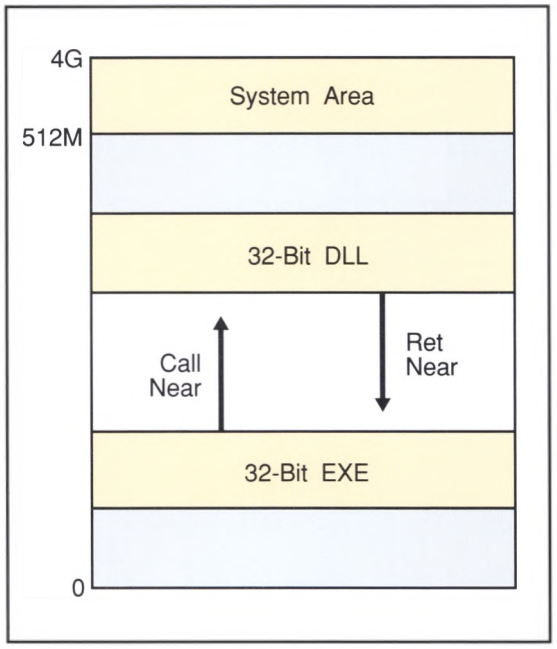


Figure 2. Flat Model DLL

OS/2 1.X COMPATIBILITY

As stated previously, OS/2 2.0 will run all OS/2 1.X application and subsystem binaries without change. To provide this, the OS/2 designers had to develop an architecture in which 16-bit and 32-bit modules could reside simultaneously. This task is particularly difficult because the segmented and flat memory models are extremely different. The major requirement for laying a foundation in which both models can coexist is a high-performance mechanism for converting 16-bit addresses to 32-bit addresses and vice versa. Once this problem is solved, the task of servicing a 16-bit API call with a 32-bit routine, or the opposite, becomes feasible. The technique used to deal with address conversions between the segmented and flat models is called *LDT tiling*.

A tiled LDT contains at most 8,192 segment descriptors, each of which maps a 64 KB region of the process address space; the regions mapped by contiguous descriptors are contiguous in the process address space. The result is that a single LDT can map, at most, 512 MB of contiguous linear address space in the flat model. This is why the OS/2 2.0 release restricts the size of the process address space to 512 MB.

This tiling of the LDT creates an address mapping between the 16:16 address and the 0:32 address for any byte of memory in the process address space. Figure 3 illustrates how 16-bit segments are allocated in the process address space and mapped into the LDT. The following formulas describe the address conversions between the 16-bit and 32-bit memory models:

0:32 = (SELECTOR(16:16)>>3)<< + OFFSET(16:16)

The 0:32 pointer is constructed from a 16:16 pointer by taking the selector, eliminating the table indicator and privilege level, putting it into the high-order 16 bits of the 0:32 pointer, and adding in the offset from the 16:16 pointer.

16:16 = MAKEP(((HIWORD(0:32))<<)+7, LOWORD(0:32))

The 16:16 pointer is constructed from a 0:32 pointer by the reverse process—take the high-order 16 bits of the 0:32 pointer, add in the privilege level and table indicator information, and that is the selector of the 16:16 pointer. The offset portion of the 16:16 pointer is the low-order 16 bits of the 0:32 pointer.

To make this address mapping work, the LDT is managed differently in OS/2 2.0 than in OS/2 1.X. In 1.X the LDT selectors for private and shared memory were interspersed in the LDT. The LDT in 2.0 is managed sparsely; private selectors are allocated from one end of the LDT while shared selectors are allocated from the other. This does not affect the compatibility of 16-bit modules, since their segmented nature makes them relocatable by default.

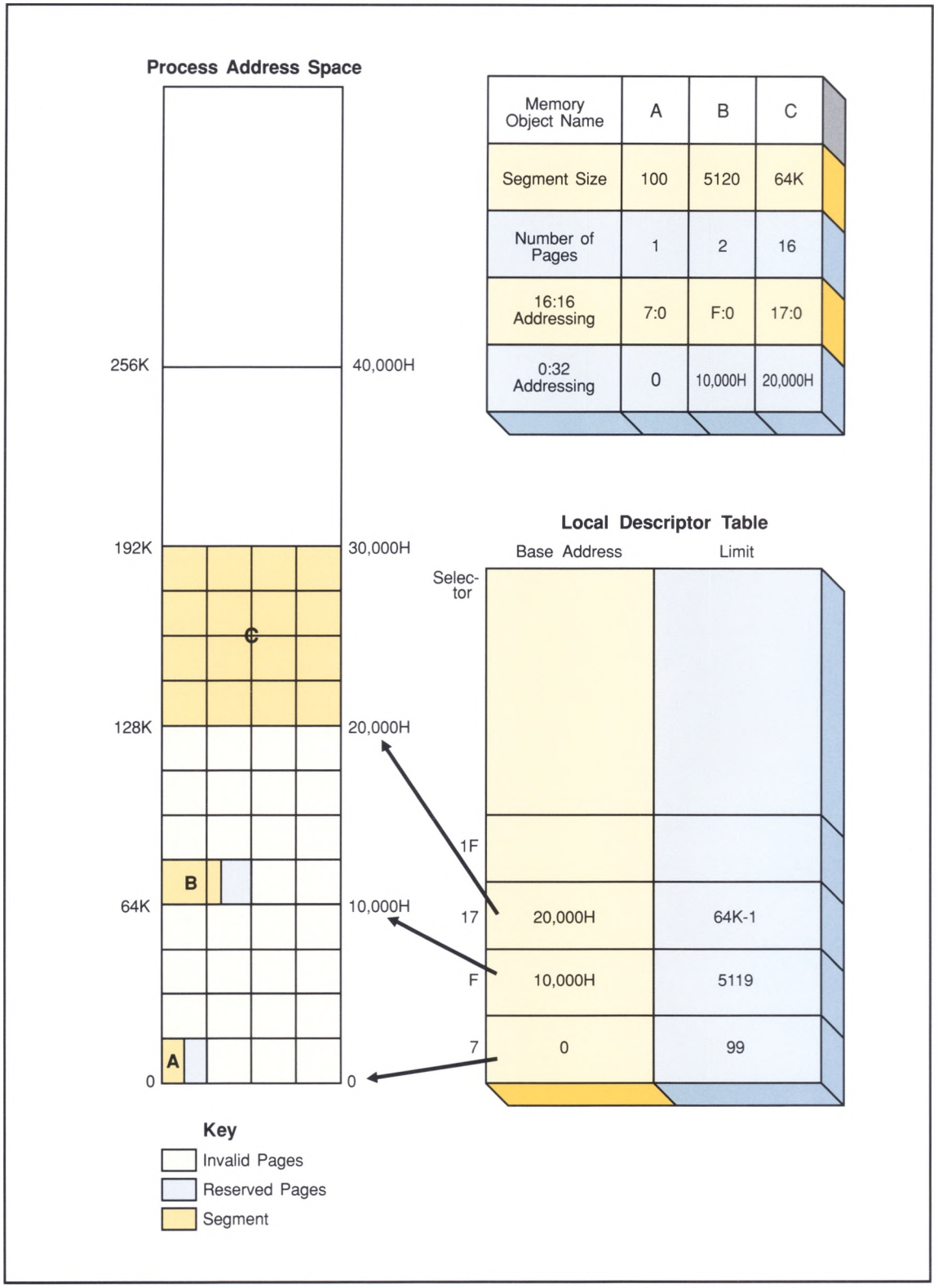


Figure 3. Tiled Address Space Next to LDT

The memory manager puts all 16-bit API memory requests on 64 KB linear boundaries and sets up the LDT to access the memory as described. This guarantees that the system will also be able to satisfy 16-bit DosReallocSeg requests up to the 64 KB maximum. Notice in Figure 3 that each 16-bit segment takes up at least one page of virtual and physical memory, as well as one page on the swap device. Therefore, 16-bit applications that create many small segments will fragment the pages that constitute the address space. More memory is consumed than when the same application is run on OS/2 1.X, although part of this memory fragmentation effect is reduced since the system swaps pages instead of segments. The trade-off is necessary, however, to provide a high-performance mechanism for address sharing between the two different memory models.

The rest of the 16-bit compatibility comes from a layer of procedures that take 16-bit API requests, convert them into 32-bit API requests, issue the requests, and complete the API return conditions with 16-bit semantics. A routine that does this for a single API is called a *thunk* (specifically, a 16-to-32 thunk). Thunks that are created to go in the opposite direction, from 32-bit semantics to 16-bit semantics, are called 32-to-16 thunks. Since the entire system is not 32-bit in the initial 2.0 release, it contains both types of thunks. Thunks are merely tools that make it possible to build one type of API (16 or 32) from the other when both APIs are needed in one system—thunks are *not* APIs.

MULTIPLE VIRTUAL DOS MACHINES

OS/2 2.0 provides DOS 4.0 compatibility using the virtual 8086 mode of the 80386 processor, and uses paging to provide more than one DOS-compatibility environment. The DOS environment offered with OS/2 2.0 is more DOS-compatible than the one offered with OS/2 1.X because it is able to encapsulate the entire DOS environment in a virtual machine. This gives the system much better protection. In OS/2 1.X, an errant DOS application could conceivably hang the entire system. In OS/2 2.0, however, an errant DOS application can only hang its own DOS session; the hung DOS session can then be terminated from the shell.

DOS applications can be run full-screen, windowed, or iconized in the background. Besides being better protected and providing better compatibility and more DOS sessions, the OS/2 2.0 DOS environment leaves applications more than 600 KB in which to execute —more than DOS! Support for both Expanded Memory Specification (EMS) and Extended Memory Specification (XMS) is provided. DOS asynchronous communications applications can support communication speeds of up to 9600 baud. Since the DOS environments are swappable, starting many DOS sessions will not increase system memory requirements.

The DOS environment in OS/2 2.0 also provides an extendable OEM architecture that allows the environment to be custom tailored to emulate any DOS environment. At the heart of this extendability is the Virtual Device Driver (VDD) architecture. All of the low-level DOS support, which in OS/2 1.X resided in "physical" device drivers, has been moved into virtual device drivers. Because of the 80386 virtual 8086 mode, all interrupt processing is done in protect mode; bimodal device drivers are no longer needed. The new driver architecture provides Physical Device Drivers (PDDs) for basic device support and VDDs for supporting virtual devices to the DOS environments.

OS/2 2.0 provides DOS 4.0 compatibility using the virtual 8086 mode of the 80386 processor.

DEVICE DRIVER MODEL

The device driver model is still 16 bits, but it has been reorganized into an architecture that separates the DOS support from the basic OS/2 device support. Most OS/2 1.X device drivers work "as is" unless they provide support for the DOS environment (using the DevHelp function SetROMVector). In these cases the driver still works, but not for the DOS environment. The code written in these drivers as bimodal code still works in OS/2 2.0, but is only called in protect mode.

OS/2 2.0's basic device driver model has also been enhanced to support devices in a paged environment. Most devices that use DMA cannot deal with blocks of noncontiguous memory, which occur as soon as paging is enabled. Therefore, when a device driver locks down memory to do I/O, it should be aware of the device's capabilities (or incapabilities) for dealing with a paged environ-

ment. One could say that DMA comes underneath the paging model. To provide better support for paged environments, many devices that use DMA will support a feature called Scatter/Gather. Scatter/Gather is an I/O protocol used by a device to do DMA to and from noncontiguous memory regions. Scatter/Gather is typically found on bus master and high-performance/high-bandwidth devices.

The Device Help (DevHelp) routines have been enhanced to provide page locking (instead of segment locking) and generation of Scatter/Gather lists.

To provide better support for paged environments, many devices that use DMA will support a feature called Scatter/Gather.

At COMDEX/Fall, IBM and Microsoft announced their intention to deliver a 32-bit version of OS/2 in 1990 (see the Winter '90 issue of the *Developer*, pages 4–9). They also made available a jointly-developed Software Development Kit (SDK) to assist developers in preparing for the new 32-bit OS/2 (Winter issue, pages 103–104). As we went to press, no formal announcement (official product name, content, availability, and so on) has been made by the two companies. This and other articles in this issue discuss functions that are currently provided only in the SDK. It is possible that some features in the SDK may not be included in the final OS/2 product, just as some additional features may appear in the OS/2 product that are not in the SDK today.

Michael S. Kogan, *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, FL 33432.*
Mr. Kogan is a staff programmer currently on leave from the IBM OS/2 Design Staff to complete a Doctorate of Science (ScD) in Computer Science at Nova University in Fort Lauderdale, Florida. In 1984, he joined IBM in Boca Raton, where he developed and tested several products in the IBM Engineering/Scientific Software Series. He then participated in the design, development, and testing of the IBM Personal Computer XENIX implementation. He spent over two years as a developer of 16-bit OS/2 and another two and a half years as a lead designer of 32-bit OS/2. Mr. Kogan holds a BS in Computer Science and Mathematics from Emory University in Atlanta, Georgia, and an MS in Computer Science from Nova University.

OS/2 2.0 32-Bit API

by Michael S. Kogan

The OS/2 2.0 system provides a new 32-bit Application Programming Interface (API) to allow access to the system APIs by flat model programs and subsystems. This article illustrates the differences in the OS/2 1.X and the OS/2 2.0 APIs for most of the base system components.

The OS/2 2.0 32-bit API contains functions analogous to those found in the OS/2 1.X API. These functions, however, use the flat model near-calling conventions and 0:32 pointer types, and allow the system API to access memory objects larger than 64 KB. The 32-bit Presentation Manager (PM) APIs are analogous to the PM APIs in OS.2 1.X systems, but the other base APIs that come with the OS/2 Standard Edition (OS/2 SE) have been changed to provide a more consistent and coherent API for programmers. Although most of the same base system functionality exists, some of the API names have changed, and some redundant APIs have been eliminated.

Although the Keyboard (KBD), Video (VIO), Mouse (MOU), and Monitor (MON) components do not have analogous 32-bit APIs, the 16-bit versions of these APIs may be used by 32-bit mixed applications (see article in this issue entitled "OS/2 2.0 Tools and Program Development" describing application models). In the tables that follow, N/A (Not Applicable) implies that the API does not exist. They allow the developer to identify differing functionality between the 16-bit and 32-bit APIs.

TASKING

The tasking component of OS/2 2.0 has the same functionality as in OS/2 1.X except that the system supports up to 4K threads. Also, the tasking APIs have been modified to provide better thread control.

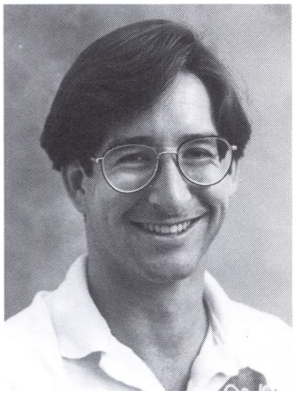
DosCreateThread has been enhanced to allow dynamic thread stack growth, system management of the stack, and creation of suspended threads. A new function, the DosWaitThread API, allows a thread to block waiting on other threads within a process to terminate. The DosDebug API is used to support 32-bit debuggers instead of DosPTrace.

The InfoSeg architecture of OS/2 1.X has been redefined for the flat model environment. The static values from the local and global InfoSegs are now returned by the DosQuerySysInfo API. The per-thread and per-process variables are in a new data structure called the Thread Information Block (TIB). A thread may gain access to its TIB by calling the DosGetThreadInfo API. The DosGetPrty, DosGetPID, and DosGetPPID APIs no longer exist, since this information is available in the TIB.

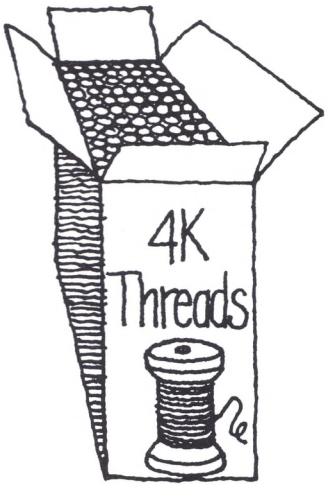
Since DosR2StackRealloc, DosCallBack, and DosRetForward do not make sense in the 32-bit flat environment, there are no 32-bit versions of these functions.

SIGNALS

Signals have been combined into the exception management architecture. The CTRL+C, CTRL+BREAK, and KILLPROCESS signals have been defined as exceptions. See the



Michael S. Kogan



32

TASKING APIs	
16-Bit Name	**32-Bit Name**
DosCreateThread	DosCreateThread
DosCWait	DosWaitChild
N/A	DosWaitThread
DosEnterCritSec	DosEnterCritSec
DosExecPgm	DosExecPgm
DosExit	DosExit
DosExitCritSec	DosExitCritSec
DosExitList	DosExitList
DosGetInfoSeg	N/A
N/A	DosGetThreadInfo
DosGetPrty	N/A
DosKillProcess	DosKillProcess
DosSetPrty	DosSetPriority
DosGetPID	N/A
DosGetPPID	N/A
DosR2StackRealloc	N/A
DosCallBack	N/A
DosRetForward	N/A
N/A	DosDebug
DosPTrace	N/A
DosResumeThread	DosResumeThread
DosSuspendThread	DosSuspendThread
SIGNAL APIs	
DosHoldSignal	N/A
DosSetSigHandler	N/A
DosSendSignal	N/A
ANONYMOUS PIPE API	
DosMakePipe	DosCreatePipe
QUEUEING APIs	
DosCreateQueue	DosCreateQueue
DosOpenQueue	DosOpenQueue
DosCloseQueue	DosCloseQueue
DosPeekQueue	DosPeekQueue
DosPurgeQueue	DosPurgeQueue
DosQueryQueue	DosQueryQueue
DosReadQueue	DosReadQueue
DosWriteQueue	DosWriteQueue

SEMAPHORE APIs	
16-Bit Name	**32-Bit Name**
DosSemClear	N/A
DosSemRequest	N/A
DosSemSet	N/A
DosSemSetWait	N/A
DosSemWait	N/A
DosMuxSemWait	N/A
DosCreateSem	N/A
DosOpenSem	N/A
DosCloseSem	N/A
DosFSRamSemRequest	N/A
DosFSRamSemClear	N/A
N/A	DosCreateMutexSem
N/A	DosOpenMutexSem
N/A	DosCloseMutexSem
N/A	DosRequestMutexSem
N/A	DosReleaseMutexSem
N/A	DosQueryMutexSem
N/A	DosCreateEventSem
N/A	DosOpenEventSem
N/A	DosCloseEventSem
N/A	DosResetEventSem
N/A	DosPostEventSem
N/A	DosWaitEventSem
N/A	DosQueryEventSem
N/A	DosCreateMuxWaitSem
N/A	DosOpenMuxWaitSem
N/A	DosCloseMuxWaitSem
N/A	DosWaitMuxWaitSem
N/A	DosAddMuxWaitSem
N/A	DosDeleteMuxWaitSem
N/A	DosQueryMuxWaitSem
NAMED PIPE APIs	
DosCallNmPipe	DosCallNPipe
DosConnectNmPipe	DosConnectNPipe
DosDisConnectNmPipe	DosDisConnectNPipe
DosMakeNmPipe	DosCreateNPipe
DosPeekNmPipe	DosPeekNPipe
DosQNmPHandState	DosQueryNPHState
DosQNmPipeInfo	DosQueryNPipeInfo
DosQNmPipeSemState	DosQueryNPipeSemState

32

NAMED PIPE APIs	
16-Bit Name	**32-Bit Name**
DosRawReadNmPipe	DosRawReadNPipe
DosRawWriteNmPipe	DosRawWriteNPipe
DosSetNmPHandInfo	DosSetNPHState
DosSetNmPipeSem	DosSetNPipeSem
DosTransactNmPipe	DosTransactNPipe
DosWaitNmPipe	DosWaitNPipe
TIMER SERVICES APIs	
DosGetDateTime	DosGetDateTime
DosSetDateTime	DosSetDateTime
DosSleep	DosSleep
DosTimerAsync	DosAsyncTimer
DosTimerStart	DosStartTimer
DosTimerStop	DosStopTimer
MEMORY MANAGEMENT APIs	
DosAllocSeg	N/A
DosAllocShrSeg	N/A
DosGetShrSeg	N/A
DosGetSeg	N/A
DosGiveSeg	N/A
DosReallocSeg	N/A
DosFreeSeg	N/A
DosAllocHuge	N/A
DosGetHugeShift	N/A
DosReallocHuge	N/A
DosCreateCSAlias	N/A
DosLockSeg	N/A
DosUnLockSeg	N/A
DosMemAvail	N/A
DosSizeSeg	N/A
DosGetResource	DosGetResource
DosSubAlloc	DosSubAlloc
DosSubFree	DosSubFree
DosSubSet	DosSubSet
N/A	DosSubUnset
N/A	DosAllocMem
N/A	DosAllocSharedMem
N/A	DosGetNamedSharedMem
N/A	DosGetSharedMem
N/A	DosGiveSharedMem

MEMORY MANAGEMENT APIs	
16-Bit Name	**32-Bit Name**
N/A	DosFreeMem
N/A	DosSetMem
N/A	DosQueryMem
DYNAMIC LINKING APIs	
DosLoadModule	DosLoadModule
DosFreeModule	DosFreeModule
DosGetProcAddr	DosQueryProcAddr
DosGetModHandle	DosQueryModuleHandle
DosGetModName	DosQueryModuleName
DosQAppType	DosQueryAppType
DosGetMachineMode	N/A
BadDynLink	N/A
DosGetVersion	N/A
DosGetEnv	N/A
DEVICE I/O APIs	
DosBeep	DosBeep
DosCLIAccess	N/A
DosPortAccess	N/A
DosDevConfig	DosDevConfig
DosPhysicalDisk	DosPhysicalDisk
FILE SYSTEM APIs	
DosBufReset	DosResetBuffer
DosChDir	DosSetCurrentDir
DosChgFilePtr	DosSetFilePtr
DosClose	DosClose
DosDelete	DosDelete
DosDevIOCTL	DosDevIOCTL
DosDupHandle	DosDupHandle
DosEditName	DosEditName
DosFileIO	DosFileIO
DosFileLocks	DosSetFileLocks
DosFindClose	DosFindClose
DosFindFirst	DosFindFirst
DosFindNext	DosFindNext
DosFindNotifyClose	DosFindNotifyClose
DosFindNotifyFirst	DosFindNotifyFirst
DosFindNotifyNext	DosFindNotifyNext
DosFSAttach	DosFSAttach
DosFSCtl	DosFSCtl

32

FILE SYSTEM APIs	
16-Bit Name	**32-Bit Name**
DosMkDir	DosCreateDir
DosMove	DosMove
DosNewSize	DosSetFileSize
DosOpen	DosOpen
DosQCurDir	DosQueryCurrentDir
DosQCurDisk	DosQueryCurrentDisk
DosQFHandState	DosQueryFHState
DosQFileInfo	DosQueryFileInfo
DosQFileMode	DosQueryFileMode
DosQFSAttach	DosQueryFSAttach
DosQFSInfo	DosQueryFSInfo
DosQHandType	DosQueryHType
DosQPathInfo	DosQueryPathInfo
DosQSysInfo	DosQuerySysInfo
DosQVerify	DosQueryVerify
DosRead	DosRead
DosReadAsync	N/A
DosRmDir	DosDeleteDir
DosScanEnv	DosScanEnv
DosSearchPath	DosSearchPath
DosSelectDisk	DosSetDefaultDisk
DosSetFHandState	DosSetFHState
DosSetFileInfo	DosSetFileInfo
DosSetFileMode	DosSetFileMode
DosSetFSInfo	DosSetFSInfo
DosSetMaxFH	DosSetMaxFH
DosSetPathInfo	DosSetPathInfo
DosSetVerify	DosSetVerify
DosWrite	DosWrite
DosWriteAsync	N/A
MESSAGE RETRIEVER APIs	
DosGetMessage	DosGetMessage
DosInsMessage	DosInsertMessage
DosPutMessage	DosPutMessage

CODE PAGE SUPPORT APIs	
16-Bit Name	**32-Bit Name**
DosSetCp	N/A
DosSetProcCp	DosSetProcessCp
DosGetCp	DosQueryCp
DosGetCtryInfo	DosQueryCtryInfo
DosCaseMap	DosMapCase
DosGetDBCSEv	DosQueryDBCSEnv
DosGetCollate	DosQueryCollate
SESSION MANAGEMENT APIs	
DosStartSession	DosStartSession
DosSetSession	DosSetSession
DosSelectSession	DosSelectSession
DosStopSession	DosStopSession
DosSMRegisterDD	N/A
ERROR MANAGEMENT APIs	
DosErrClass	DosErrClass
DosError	DosError
EXCEPTION MANAGEMENT APIs	
DosSetVec	N/A
N/A	DosSetExceptionHandler
N/A	DosUnsetExceptionHandler
N/A	DosRaiseException
N/A	DosUnwindException
N/A	DosSetSignalExceptionFocus

Exception Management APIs shown later in this article.

QUEUEING

The 32-bit queueing APIs have the same functionality as the 16-bit queueing APIs, although they allow 0:32 addresses and element sizes greater than 64 KB. A 32-bit application calling the 32-bit DosOpenQueue on a queue that was created by a 16-bit application calling the 16-bit DosCreateQueue is not supported. This avoids problems when mixing environments.

SEMAPHORES

The semaphore APIs have been redesigned to provide a semaphore model that is both portable to other architectures and easier to use than the original OS/2 1.X semaphore APIs. The 32-bit semaphore model does not provide an analog of a RAM semaphore because RAM semaphores are not portable to other architectures.

OS/2 2.0 provides two classes of semaphores: *private* and *shared*. A process may contain up to 64 KB private semaphores. They are used among threads within a process. Private semaphore handles are valid only in the context of the process that owns them. Shared semaphores are shared among all processes;

up to 64 KB shared semaphores may exist in the system. The origin of the threads that will need to access the semaphore determines whether private or shared semaphores are selected. If only threads within a process need access, then a private semaphore is best. Semaphores need not be named, although an optional name may be provided when a semaphore is created.

Besides being shared or private, semaphores may also assume one of three characteristics that determine the appropriate semaphore API to use. Mutex semaphores provide mutually exclusive access to a resource and use the MutexSem APIs. Event semaphores signal events and use the EventSem APIs. MuxWait semaphores can be used for waiting on multiple events or multiple mutex semaphores, and use the MuxWaitSem APIs.

MEMORY MANAGEMENT

The memory management APIs no longer deal with segments. Instead, they deal with memory objects that can be larger than 64 KB. Most functions offered in the OS/2 1.X memory management API are still available in a flat form, along with several new APIs for controlling and querying page attributes. The DosSub APIs provide support for suballocating from memory objects larger than 64 KB, as well as support for sparse heaps and optional thread serialization control.

DYNAMIC LINKING

Since the 32-bit environment provides no analog to Family API, DosGetMachineMode and BadDynLink are no longer necessary. DosGetVersion and DosGetEnv have been dropped, since this information is now in the Thread Information Block.

FILE SYSTEM

Except for some naming changes, the file system APIs are the same as in OS/2 1.X. The asynchronous I/O APIs DosReadAsync and DosWriteAsync have been dropped, since applications *can* allocate their own threads for asynchronous I/O. DosRead and DosWrite have been enhanced to support I/O count sizes of greater than 64 KB.

EXCEPTION MANAGEMENT

The exception management APIs have been redesigned to be machine independent and to allow exceptions to be handled on a per-thread instead of per-process basis. The kernel still supports the 16-bit DosSetVec API and exceptions. The 32-bit exceptions, however, are managed using the TIB, and divided into system exceptions and user-defined exceptions. The 16-bit signals and support for handling protection faults have been integrated into the exception architecture. A guard page exception has been defined to support dynamic stack growth. A process termination exception that is delivered to all threads in a process, when a process undergoes forced or voluntary termination, has also been added. Language runtimes can use the DosUnwindException API to better support non-local GOTOs, such as longjmp/setjmp in the C runtime environment. The DosSetSignalExceptionFocus is similar to the 16-bit DosSetSigHandler API. It notifies the system that the process is prepared to handle the CTRL-C and CTRL-BREAK signal exceptions.

Michael S. Kogan, *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, FL 33432. Mr. Kogan is a staff programmer currently on leave from the IBM OS/2 Design Staff to complete a Doctorate of Science (ScD) in Computer Science at Nova University in Fort Lauderdale, Florida. In 1984, he joined IBM in Boca Raton, where he developed and tested several products in the IBM Engineering/Scientific Software Series. He then participated in the design, development, and testing of the IBM Personal Computer XENIX implementation. He spent over two years as a developer of 16-bit OS/2 and another two and a half years as a lead designer of 32-bit OS/2. Mr. Kogan holds a BS in Computer Science and Mathematics from Emory University in Atlanta, Georgia, and an MS in Computer Science from Nova University.*

32-Bit OS/2

OS/2 2.0 Tools and Program Development

by Michael S. Kogan and Robert Tycast

To facilitate 32-bit program development for the OS/2 2.0 system, the OS/2 toolkit has been enhanced to contain a 32-bit C compiler, macro assembler, link editor, system libraries, and Include files. These tools allow programmers to take advantage of the flat memory model and the OS/2 32-bit API. The 32-bit OS/2 programming environment provides only one memory model for applications, unlike the 16-bit OS/2 environment, which provides several different models that allow different degrees of segmentation to be controlled in the program.

APPLICATION MODELS

Since OS/2 2.0 supports both 16-bit and 32-bit Application Programming Interfaces (APIs), the application models available to the programmer support varying degrees of API mixing. A 32-bit application that needs some 16-bit APIs is likely since some applications can benefit greatly from the flat memory model, but depend on some subsystem that currently does not provide a 32-bit API. Therefore, in the OS/2 2.0 environment, applications are classified according to their user I/O model and also by their API usage.

The following classes describe applications according to their user I/O model:

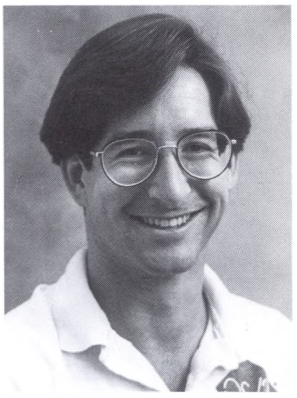
Full-Screen Applications: Full-screen applications use the Keyboard (KBD), Video (VIO), and Mouse (MOU) APIs to perform their user I/O. Some full-screen applications take complete control of the video buffer management. These applications cannot be run in a Presentation Manager (PM) window. Most applications in this category can be run in a PM window and use a combination of C runtime interfaces and VIO APIs to provide a fast, text-based user interface.

Window-Compatible Applications: Window-compatible applications include most full-screen applications that do not take control of the video buffer (use VIOGETPHYSBUF), or those that use the DOSREAD/DOSWRITE functionality to the standard I/O handles. This category includes most C programs that use printf and the C runtime library text output and keyboard input functions. Applications of this type can run in a PM window, but do not directly take advantage of windows, icons, or the mouse and screen functions that come with PM.

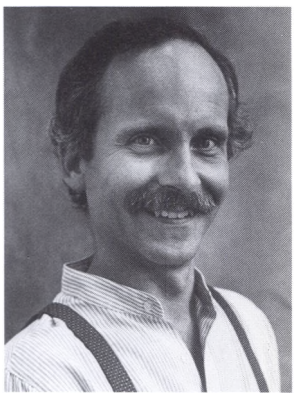
Presentation Manager Applications: PM applications use the message-based input queue model and take advantage of the complete PM screen and mouse functionality. PM applications provide the user with a consistent user interface and a What-You-See-Is-What-You-Get (WYSIWYG) view of their screen data.

The application models available on OS/2 2.0 are categorized from a 16-bit and 32-bit standpoint. The OS/2 2.0 32-bit API provides 32-bit versions of most APIs present in the 16-bit API, but does not include 32-bit VIO, KBD, or MOU APIs. This is because the PM interface is the strategic application user I/O programming interface that will be carried into the future. The VIO/KBD/MOU functionality will still be present in the 16-bit API, and can be used by a 32-bit application on 80386-based OS/2 2.X systems.

16-Bit Pure Applications: These applications are built using the 16-bit language tools and linker (LINK.EXE). All three application classes are available in this model. It is possible for applications in this class to use some 32-bit instructions on 80386-based computers.



Michael S. Kogan



Robert Tycast

Since the 16-bit OBJ format and 16-bit linker have certain limitations, it is not possible to call 32-bit APIs or have 0:32 fixups in this model. Applications in this category run "as is" on 80386-based OS/2 2.0; they also will run on OS/2 1.X systems. The 16-bit pure applications do not benefit from the flat memory model or have the portability offered by the pure 32-bit programming model. See Figure 1 for a 16-bit pure applications model.

16-Bit Mixed Applications: The 16-bit and/or 32-bit language tools and the 32-bit linker (LINK386.EXE) are used to build this type of application. All three application classes are available in this model. Since the application is built with the 32-bit linker, applications in this model run only on 80386-based OS/2 2.X systems. Although they are predominantly 16-bit, 32-bit instructions and 32-bit APIs may be used. Since these applications are in the 32-bit EXE format, they benefit from the demand paging of OS/2 2.0, but do not take advantage of the flat memory model or have the portability offered by the pure 32-bit programming model. See Figure 2 for a 16-bit mixed applications model.

32-Bit Pure Applications: These applications are built using the 32-bit language tools and

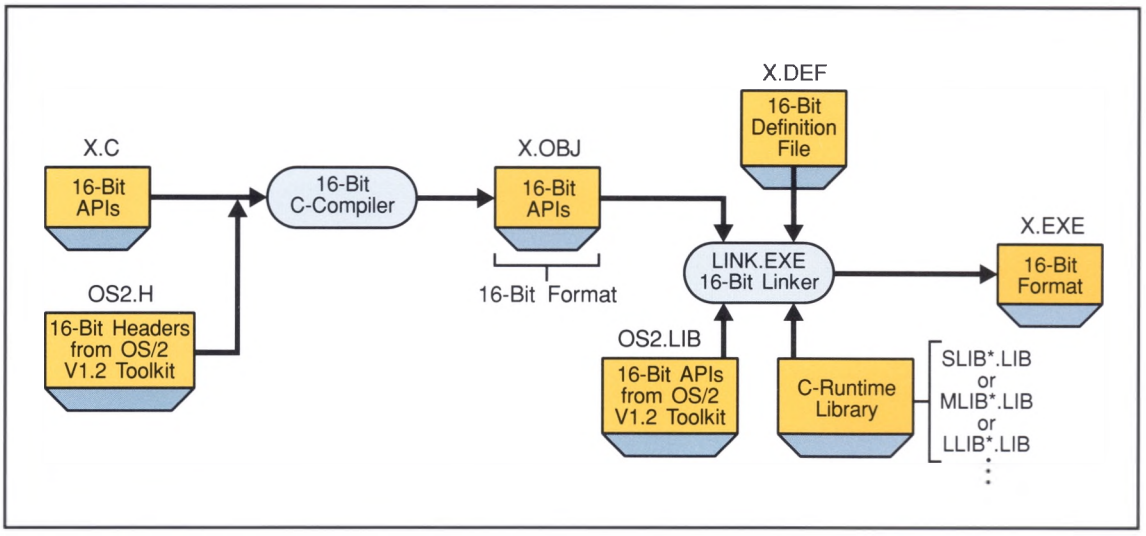


Figure 1. 16-Bit Pure Applications Model

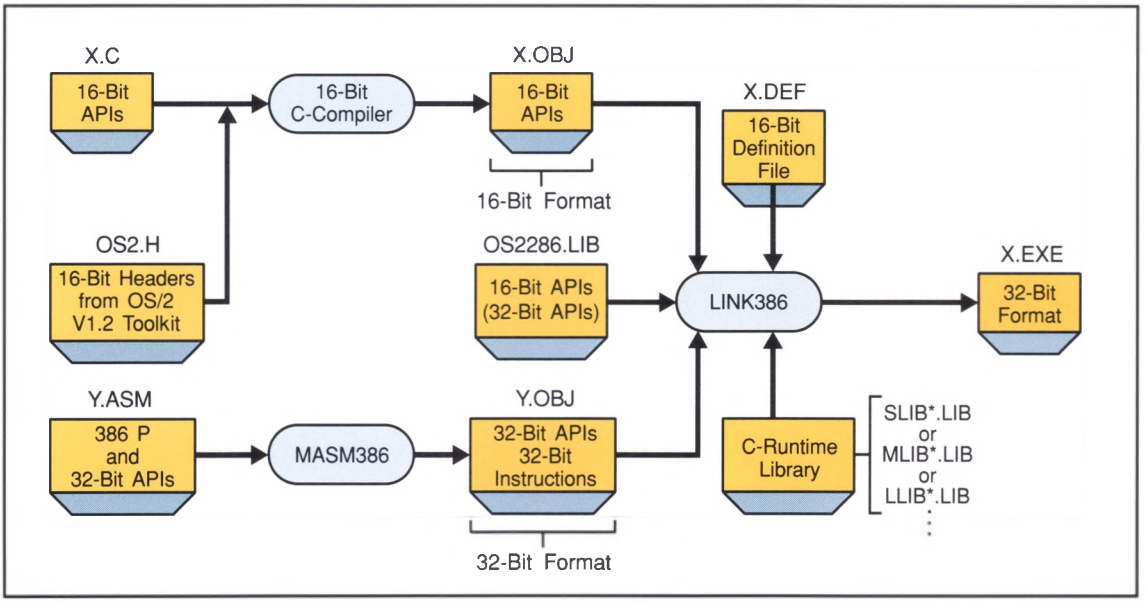


Figure 2. 16-Bit Mixed Applications Model

the 32-bit linker (LINK386.EXE). Only the PM and Window-compatible application classes are available in this model. Applications in this model use only the 32-bit API and are source portable to any future 32-bit OS/2 system. Note that applications of this model may not use any VIO/KBD/MOU APIs since these are not in the 32-bit API. These applications use the flat memory model and gain the full benefit of 32-bit application execution. See Figure 3 for a 32-bit pure applications model.

32-Bit Mixed Applications: These applications are the same as 32-bit pure applications, but may also call 16-bit APIs. Since this application model can call any 16-bit API, all three application classes are available, including the full-screen class. Applications in this model, however, do not benefit from the source portability found in 32-bit pure applications. See Figure 4 for a 32-bit mixed applications model.

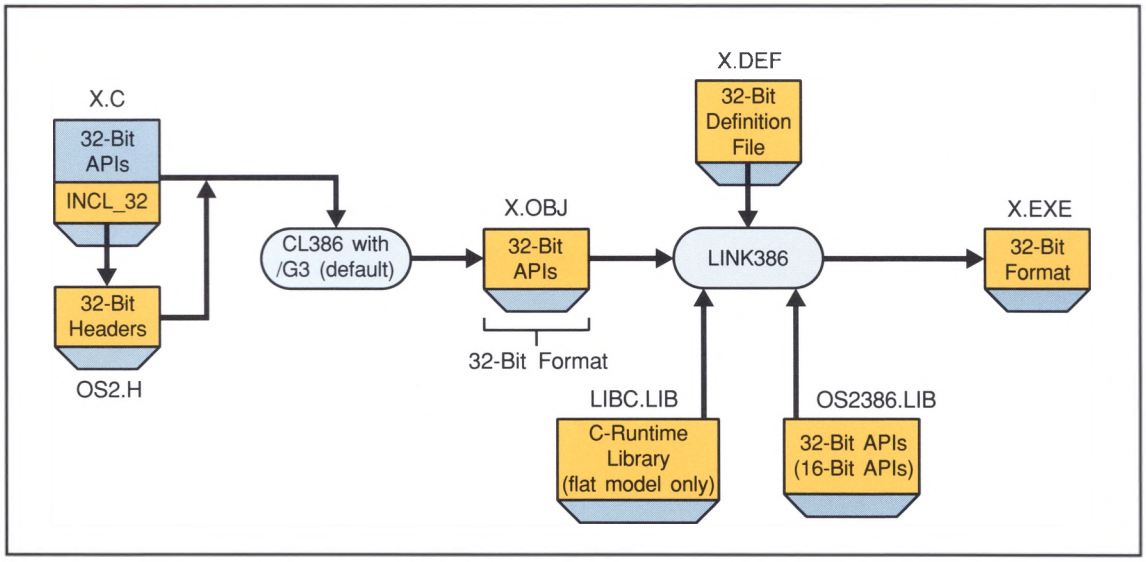


Figure 3. 32-Bit Pure Applications Model

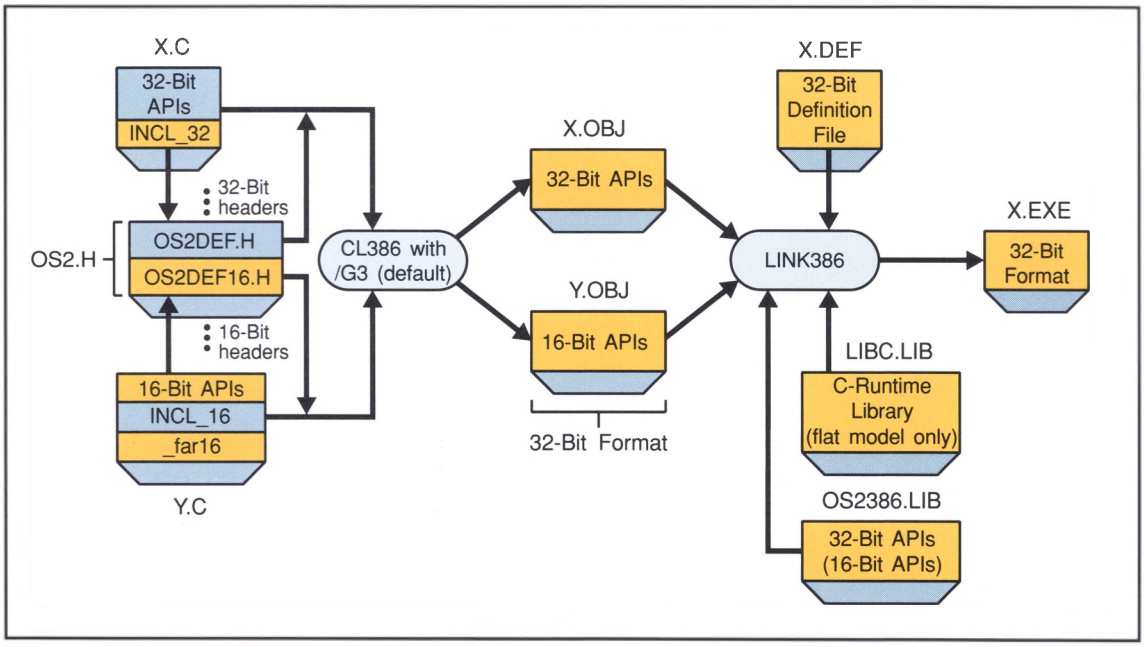


Figure 4. 32-Bit Mixed Applications Model

APPLICATION MIGRATION

Developer's ask, "Which model should I use?" The answer depends on the code base from which the application is migrating, and on both the short-term and long-term goals and requirements of the product.

The 32-bit pure model using PM is clearly the best model to migrate toward. With a 16-bit, non-PM application, however, one of the mixed models may be better, offering a near-term benefit from the flat memory model or some 32-bit execution. Porting a 16-bit PM application to the strategic 32-bit pure model is straightforward. For porting an application from another architecture or a larger computer, the 32-bit pure application model may be best.

The 32-bit pure model using PM is clearly the best model to migrate toward.

The amount of effort required to transfer the code to the 32-bit models depends on which 16-bit memory model the original application uses. Since 16-bit C applications may use one of several runtime memory models for dealing with segmentation, the source code will have a different flavor of segment semantics throughout ("far" keywords, for example), depending on the memory model. In general, to migrate from C code to the flat memory model environment, all segmented structures and pointers must be "flattened;" far references must be removed; and calls to the C runtime must be flat model instead of segmented. Migrating to the 32-bit OS/2 APIs is easier if the application uses the definitions in the OS2DEF.H include file. Then, the code is usually portable and easy to migrate to the 32-bit platform.

Because the 16-bit and the 32-bit APIs and memory models differ, one set of source code may not be recompilable for both systems. The goals and requirements for market coverage, performance, and portability on 16-bit and 32-bit OS/2 systems should be considered when determining the best development methodology.

PROGRAMMING TOOLS

Programming tools for OS/2 2.0 center on the two mixed application models and the 32-bit pure application model. The 16-bit pure application model is built using the tools, libraries, and include files that come with the 16-bit language products and toolkits.

MASM386: MASM386 for OS/2 2.0 supports 32-bit 80386 programming. Modules created with MASM are 32-bit OBJ modules for use with the 32-bit LINK386.EXE utility.

C Compiler: The C compiler for OS/2 2.0, called CL386, supports programming in the flat-memory model environment. It includes a flat model C runtime library and C runtime include files. The /G3 switch is used for 32-bit compiles that produce 32-bit OBJ modules. New keywords supported include "_sysentry" used for the new 32-bit API calling convention, and "_far16," a special keyword that supports calling 16-bit APIs from 32-bit modules and 16:16-0:32 pointer conversions for the 32-bit mixed application model.

CodeView: The CodeView debugger supports development and debugging of applications in the 32-bit EXE format. It is essentially the same as the 16-bit debugger, enhanced to deal with the 32-bit environment.

LINK386.EXE: The 32-bit linker accepts 16-bit and 32-bit OBJ files. Unlike the 16-bit LINK.EXE, however, which emits both 16-bit OS/2 and DOS EXE formats, LINK386.EXE emits only 32-bit EXE modules.

System Libraries: The 16-bit OS/2 toolkit contains only one system library, called OS2.LIB. This has been replaced by two libraries called OS2286.LIB and OS2386.LIB in the 32-bit OS/2 toolkit. The OS2286.LIB library links 16-bit mixed applications, whereas the OS2386.LIB library links 32-bit pure and 32-bit mixed applications.

Include Files: The OS2.H include files support calling the 32-bit APIs from C modules using the INCL_32 include switch. A switch called INCL_16 allows a 32-bit C module to call 16-bit APIs. The include files are set up to allow a single C module to call either 16-bit or 32-bit APIs, but not both. Two C modules that call the different APIs can still be linked into a single EXE.

During a 32-bit compile with the /G3 switch, the 32-bit API is included by default using the INCL_32 symbolic definition. If the compile uses the /G3 switch and the INCL_16 symbol is defined, then the 32-bit callable 16-bit OS/2 1.X API is included. Compiling with the 16-bit compiler using the /G2 switch causes the 16-bit OS/2 1.X API to be included in a 16-bit module.

CONCLUSIONS

The OS/2 2.0 programming platform offers a variety of application classes and models to facilitate development and migration of applications to the 32-bit OS/2 platform. Although the 32-bit programming model is much simpler than the 16-bit model, it is complicated since both 16-bit and 32-bit applications and APIs are available in the same system. Judicious application planning and design, however, allow program developers to provide the highest performance applications for 80386-based systems, with a defined path for future application portability.

Michael S. Kogan, *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, FL 33432. Mr. Kogan is a staff programmer currently on leave from the IBM OS/2 Design Staff to complete a Doctorate of Science (ScD) in Computer Science at Nova University in Fort Lauderdale, Florida. In 1984, he joined IBM in Boca Raton, where he developed and tested several products in the IBM Engineering/Scientific Software Series. He then participated in the design, development, and testing of the IBM Personal Computer XENIX implementation. He spent over two years as a developer of 16-bit OS/2 and another two and a half years as a lead designer of 32-bit OS/2. Mr. Kogan holds a BS in Computer Science and Mathematics from Emory University in Atlanta, Georgia and an MS in Computer Science from Nova University.*

Robert Tycast, *IBM Entry Systems Division, P.O. Box 1328, Boca Raton, FL 33432. Mr. Tycast is an advisory programmer in the OS/2 Design Department. His current assignment is the OS/2 2.0 Information and Programming Tools project. Mr. Tycast came to IBM in 1989 from Digital Equipment Corporation where he served in several software development and software support capacities in the U.S., Latin America, and Europe. His project experience over the past thirteen years includes X11 windows, artificial intelligence technology (LISP and OPS5 support), and technical workstations (VMS and ULTRIX). Mr. Tycast has a BS from Massachusetts Institute of Technology.*

32-Bit OS/2

Multiple Virtual DOS Machines: A Better DOS

John G. Tyler

by John G. Tyler

OS/2 Version 1.0 in 1987 introduced the DOS compatibility mode (sometimes known as the "DOS Box") to allow users to run DOS applications beside new 16-bit OS/2 applications. Since the 16-bit DOS Mode is based on the Intel 80286 architecture, DOS applications run as real-mode tasks along the protected-mode environment of OS/2. OS/2 Version 2.0 introduces a technology called Multiple Virtual DOS Machines (MVDM), which is a totally redesigned DOS-compatibility environment for OS/2. MVDM extends the DOS-compatibility features of OS/2 by exploiting the Intel 80386's virtual 8086 mode, resulting in a more compatible, multitasking DOS-compatibility environment.

Most users considering a switch to OS/2 demand that OS/2 provide a DOS-compatible environment to run their favorite DOS applications. They also expect OS/2's multitasking capability, one of OS/2's most important features, to apply to their current DOS applications. Because OS/2 2.0 is based on the 80386 processor, these requirements are satisfied for the first time.

With a design based on the virtual 8086 mode, OS/2 2.0 maintains a protected system environment, even while running DOS applications. This enhances system integrity and makes it possible to multitask DOS applications with OS/2 applications, so DOS programs may continue to execute in the background. The OS/2 demand-paging mechanism allocates system memory for each Virtual DOS Machine (VDM); therefore, multiple DOS applications run in virtual memory without increasing the system's physical memory requirement. OS/2 also allows DOS applications to display in windows under the control of Presentation Manager (PM).

This article examines the differences between the real-mode OS/2 DOS Box and MVDM. It describes how OS/2 uses virtual 8086 mode for virtual I/O and DOS emulation. It also provides an extendable device virtualization architecture called Virtual Device Drivers (VDDs).

LIMITATIONS OF OS/2 1.X DOS COMPATIBILITY

The original design of OS/2 is based on the capabilities of the Intel 80286 processor; therefore, the only practical way to run a DOS application in 16-bit versions of OS/2 is in real mode. This is primarily because DOS applications address code and data in memory using a Segment:Offset address format, based on the earlier Intel 8088/8086 processors. In protected mode, however, all programs must use Selector:Offset address formats. OS/2 applications are written to run in protected mode, and the multitasking design allows many OS/2 applications to execute concurrently. To run a DOS application simultaneously with many OS/2 applications, the system switches between real mode and protected mode when necessary.

Performance Impact from Mode Switching

The Intel 80286 CPU can switch easily from real mode to protected mode, although this takes many CPU cycles. However, it has no provision for switching directly back to real mode. Switching back to real mode requires reinitializing the processor—in effect setting it to an initial power-on state. This must be done while preserving all the system control information, all process contexts, and system

resource states. Thus, switching frequently between real mode, (where a DOS program is executing) and protected mode (where several OS/2 programs may be running), generates a significant level of system overhead. Therefore, 16-bit OS/2 supports only a single DOS application running in the foreground.

Mode switching overhead is reduced on the Intel 80386, where a single instruction switches the system quickly from real mode to protected mode and back again. Although this saves some time, mode switching still requires significant overhead and is avoided whenever possible. For example, nearly all OS/2 1.X device driver code is bimodal. That is, the device driver is written so that it can be entered in either protected mode or real mode. An OS/2 bimodal device driver will force a mode switch only when it is absolutely necessary, such as when it is running in real mode, but it must move data to a protected-mode OS/2 application's virtual address space. When a DOS application is running while OS/2 applications run in the background, mode switching occurs. The overhead of mode switching may degrade system performance in these scenarios.

Loss of System Integrity in Real Mode

OS/2 provides each protected-mode process with its own independent address space for code and data objects. This protects the system from applications, and encapsulates each application's code and data objects within a unique address space protected from other applications. This also helps prevent errant applications from causing a system crash, since system-owned resources are accessible only by system code. This high level of protection, a major feature of OS/2 not found in DOS, improves the integrity and reliability of the system.

When OS/2 switches into real mode, a DOS application can directly access any object in memory between 0 and 1 MB, including portions of the OS/2 kernel and device drivers. A DOS program that inadvertently accessed any of the system's code or data objects below 1 MB could disrupt the system. In addition, a DOS program could directly access any I/O device hardware and cause the device to enter an unknown state. This could render the device useless in the OS/2 protected-mode environment, and cause the OS/2 device driver to fail. Problems arising from ill-behaved DOS applications might lead eventually to a system crash and the potential loss of user data.

DOS Applications Suspended in Background

In OS/2 1.X, DOS programs are suspended when the user switches from the DOS-mode session. This is primarily to avoid the overhead associated with switching between real mode and protected mode. This makes it impractical to use DOS applications for tasks ideally suited to running in the background, such as uploading and downloading files, compiling source code, formatting large text documents, or searching databases.

Whenever the OS/2 user switches from the DOS-mode session, all execution in that session, including interrupt handling, is stopped. Therefore, if the DOS application installs an interrupt handler (that prevents conflict with OS/2 device drivers), it only receives interrupts while the DOS program is running in the foreground. DOS software that is interrupt-driven does not operate correctly when the user switches to an OS/2 session.

Limited Application Space Available

The small memory space available in OS/2 1.X prevents some DOS applications from loading. OS/2 1.2 installs some of its kernel, device-driver code, and data in fixed memory below 1 MB so system functions may execute in real mode. Therefore, DOS applications have less space for their own code and data than they have under DOS. OS/2 1.2 provides over 520 KB of memory for DOS applications when using the default system configuration.

This problem is not unusual, even in a native DOS environment, since installing memory-resident DOS programs and device drivers have the same effect. DOS 4.01 offers about 570 KB free when using the default system configuration.

To provide memory beyond 640 KB, Extended Memory Specification (XMS) and Expanded Memory Specification (EMS) are used as extensions to DOS. These specifications

allow a DOS application to access physical memory above 1 MB. Although several major DOS applications have been enhanced to use these techniques, neither EMS nor XMS is supported in OS/2 1.2.

MULTIPLE VIRTUAL DOS MACHINES

MVDM is designed to exploit a major innovation of the 80386 called Virtual 8086 (V86) mode, allowing OS/2 to execute multiple DOS tasks within the 80386 protected-mode environment. In OS/2 2.0, this DOS task is a Virtual DOS Machine (VDM), which runs as a single-threaded V86 mode process. Each VDM in OS/2 2.0 is managed as a single-process session. The OS/2 scheduler controls the task switching of VDMs the same way as OS/2 application processes.

Figure 1 provides an overview of the OS/2 2.0 system structure with the MVDM kernel and Virtual Device Drivers (VDD) shown in relation to the OS/2 kernel and Physical Device Drivers (PDD). Key components of the OS/2 kernel that provide services to

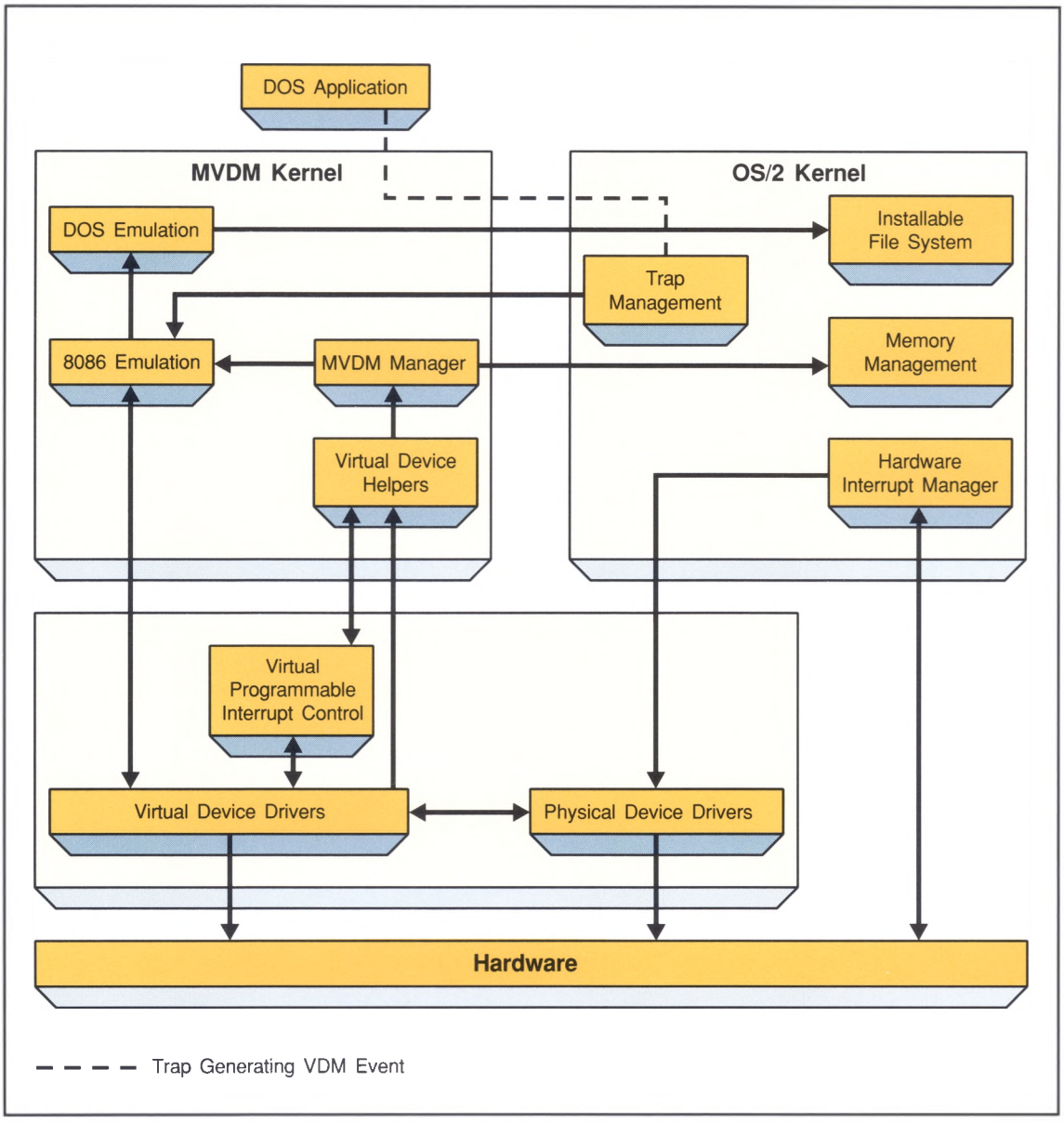


Figure 1. MVDM System Structure and Control Flow

32

MVDM TERMINOLOGY

Virtual DOS Machine (VDM): The instance of a single Virtual 8086 mode process running on an Intel 80386, 80386-SX, or 80486 processor, that is emulating all DOS operating system functions and providing a compatible environment. Several VDMs may be multitasked using the task-switching features of the Intel 80386 and compatibles.

MVDM Kernel: Composed of the following three major components, MVDM kernel controls the state and operation of up to sixteen DOS Mode sessions:

- **VDM Manager**: VDMM creates, initializes, and terminates VDMs. VDMM provides various system services to VDDs, known as Virtual Device Helper functions. VDMM manages system resources (that is, memory, timers, semaphores, files) for all VDMs.
- **8086 Emulation**: Performs 8086 instruction decoding, controls the 80386 processor's I/O Privilege Map for each VDM, manages the reflection of software interrupts for each VDM, routes IN/OUT instruction traps to the appropriate Virtual Device Drive, and manages various control structures required by each VDD.
- **DOS Emulation**: Emulates the function and operation of DOS on a per-VDM basis. Each VDM emulates an entirely independent instance of DOS. DOS services are emulated within the MVDM kernel, or by invoking protected-mode services provided by the OS/2 kernel. For example, most DOS file I/O functions are provided by the OS/2 file system.

Virtual Device Driver (VDD): A VDD emulates an aspect of system function, typically I/O. A VDD manages multiple VDMs' access to a hardware I/O devices. VDDs obtain and release system resources via the VDH services provided by the MVDM kernel. A VDD may directly access an I/O control device, or may perform I/O through a PDD using a direct call interface. A VDD may also simulate hardware interrupts into one or many VDM processes. Finally, a VDD may provide the logic to emulate BIOS and other software interrupt functions.

MVDM are shown. The major components of the MVDM kernel with a schematic view of how these components interact is also shown. With this picture as a point of reference, the following discussion highlights the advantages of MVDM and how OS/2 2.0 provides an improved DOS-compatibility environment.

Fast Mode Switching

Virtual 8086 mode is a superset of protected mode, which is enabled by setting the VM bit in the EFLAGS register of the 80386. The CPU provides high-performance hardware support to switch between V86 mode and protected mode. Because of this and other hardware support mechanisms, most of the operating system overhead associated with mode switching has been eliminated. When switching between V86 mode (DOS application) and protected mode (OS/2 application or system code), the 80386 hardware automatically activates the appropriate protection mechanisms not available on the 80286.

Real-mode execution has been totally eliminated in OS/2 2.0. This makes it possible to remove all real-mode-specific code from device drivers and kernel modules, which were bimodal executables in previous versions of OS/2. Therefore, new OS/2 2.0 device drivers are written as purely protected-mode executables. This simplifies the logic and reduces the size of the device drivers.

Improved System Integrity

By running DOS applications in virtual 8086 mode and using the paging feature of the 80386, it is possible to maintain a fully protected system environment. Each DOS program is encapsulated in its own unique 1 MB linear address space. Therefore, a DOS program cannot corrupt any system code or data, or another application's code or data. If the DOS task causes a trap or exception, it will be fully managed by the operating system to maintain integrity. An ill-behaved DOS program can be terminated cleanly by the system in much the same manner as an errant OS/2 application.

Running DOS applications in virtual 8086 mode, it is possible to maintain a fully protected system environment.

The system selectively isolates the DOS applications from I/O devices that are managed exclusively by OS/2 device drivers. These devices are then emulated, or virtualized, for one or more DOS applications. New to OS/2 2.0, the VDD provides each DOS application with a virtual instance of the real hardware, which is controlled by a PDD. If necessary for performance or other reasons, a VDD-PDD can cooperate to give the DOS program direct access to a particular I/O port or range of ports. By controlling hardware access in this way, DOS applications cannot corrupt devices that are performing I/O functions for an OS/2 process.

Multiple DOS Sessions

MVDM makes it possible to start up to sixteen concurrent DOS sessions, each operating in its own independent 1 MB linear address space. This brings true multiprogramming to the OS/2 DOS-compatibility environment. The user can run multiple DOS programs in much the same way as running multiple OS/2 applications. DOS and OS/2 applications are started in all the same ways: from a desktop group window, from the File Manager's list of files, from the DOS command prompt, and from the OS/2 START command.

The available base memory in each Virtual DOS Machine is over 610 KB.

Transfer of Data Between DOS Sessions

To allow easy migration of data from DOS applications to PM applications, OS/2 2.0 allows DOS programs to run within windows in the PM desktop environment. This allows users to display a graph or chart from a DOS application beside a document they are preparing with an OS/2 word processor. A block of text can be selected in one window (a popular DOS word processor, for example) and copied to the PM clipboard using a mouse or keyboard. That text may be pasted into another window (perhaps a Presentation Manager spreadsheet application). Graphic images can also be transferred from a DOS application to the PM clipboard. Most users' productivity can be immediately enhanced by running existing DOS programs in the windowed, multitasking environment of OS/2 2.0.

Increased Available Memory

The available base memory (below 640 KB) in each Virtual DOS Machine is over 610 KB, before loading any user-installed DOS device drivers and/or Terminate-and-Stay-Resident (TSR) programs. This increases the available base memory in OS/2 2.0 by nearly 80 KB over OS/2 1.2. Since the available base memory is greater than in DOS itself, OS/2 users are now able to load some DOS TSR programs with larger DOS applications, which would not fit together in the smaller base memory available in DOS.

Adding LAN drivers to the OS/2 configuration to support the network server or redirector functions does not take up DOS application space. Any user-installed OS/2 device drivers will not affect the amount of application space available to a DOS application running under MVDM.

EMS and XMS Memory Services

Many popular DOS applications are now using EMS and/or XMS to gain access to memory in protected mode on the 80286 or 80386 processors. This allows DOS applications to access memory above the 1 MB real-mode addressing limit, to have total code and data space larger than the available base memory, and to have very large code or data objects loaded into memory for enhanced execution speed. The standard configuration of OS/2 2.0 provides both the EMS and XMS functions.

EMS, based on the Lotus-Intel-Microsoft (LIM) EMS Version 4.0, allows DOS applications to allocate and access up to 32 MB of expanded memory in up to 255 EMS objects. These objects may be mapped into the base memory space (below 1 MB) so the DOS application may access very large address spaces. The underlying OS/2 virtual memory management functions emulate the expanded memory using the 80386 paging mechanism. Therefore, use of EMS in one VDM does not affect the ability of DOS applications in another VDM to perform similar EMS functions.

XMS functions provide another mechanism for DOS applications to access memory above the 1 MB limit. XMS supports use of the

"HIMEM" area—a 64 KB area just above 1 MB—which may be used for code or data objects by the DOS application. Other XMS functions allow moving code and data objects into extended memory, and from extended memory to base memory. Using these functions, an application may provide several overlays, where one overlay at a time is accessible to the application.

Under MVDM, EMS and XMS memory allocations are managed as OS/2 pageable virtual memory, not as fixed physical memory. Therefore, the total memory allocated can exceed the amount of system RAM.

INSIDE MVDM

The *Intel 80386 Programmer's Reference* (1986) describes the system software that supervises Virtual 8086 machines as the "V86 Monitor." To provide a V86 Monitor in OS/2, and to do this in a very clean manner that would exploit the existing services of the OS/2 kernel (task scheduling, memory management, interprocess communication, and so on), it was necessary to extend the OS/2 kernel itself. The MVDM kernel (Figure 1) is comprised of three new modules: DOS Emulation, 8086 Emulation, and the VDM Manager. These provide a full set of control program interfaces known as Virtual Device Helper (VDH) services, which are invoked by VDD modules. VDDs in turn provide device-specific support, such as hardware virtualization and BIOS emulation. Some VDDs also provide low-level system functions for other VDDs.

The VDD model is of interest to developers whose DOS applications provide hardware-specific functions on non-IBM option adapters. The following section describes the VDD architecture and gives an overview of MVDM operations with emphasis on the behavior of VDDs.

Virtual Device Helper Services

Several categories of VDH services are available to all VDDs. These services control access to system resources, such as memory, timers, and semaphores. VDM events, such as VDM foreground and background state switching, may be tracked using VDH services. A VDD may display error messages via the VDHPopUp service. Other VDH services allow a VDD to establish communications with another VDD or with a PDD. Typically, the VDD also must hook a set of system traps that are generated when certain 8086 instructions are executed in V86 mode. To get control, a VDD usually hooks the following traps using the appropriate VDH services:

- Traps generated when a DOS application performs direct I/O to a port that is being virtualized (IN and OUT instructions)
- Traps generated when a BIOS or other software interrupt function is executed in a VDM (INT instructions)
- Traps generated when a DOS application attempts to access memory that is mapped to a physical device over which the VDD has control

VDDs are written as 32-bit, flat memory model, protected-mode modules.

Another important category of VDH services are those that support the simulation of hardware interrupts. These services control the state of the virtual programmable interrupt controller. Altogether, approximately one hundred VDH services are provided. All are based on 32-bit interface standards and are callable either from assembler or C language sources.

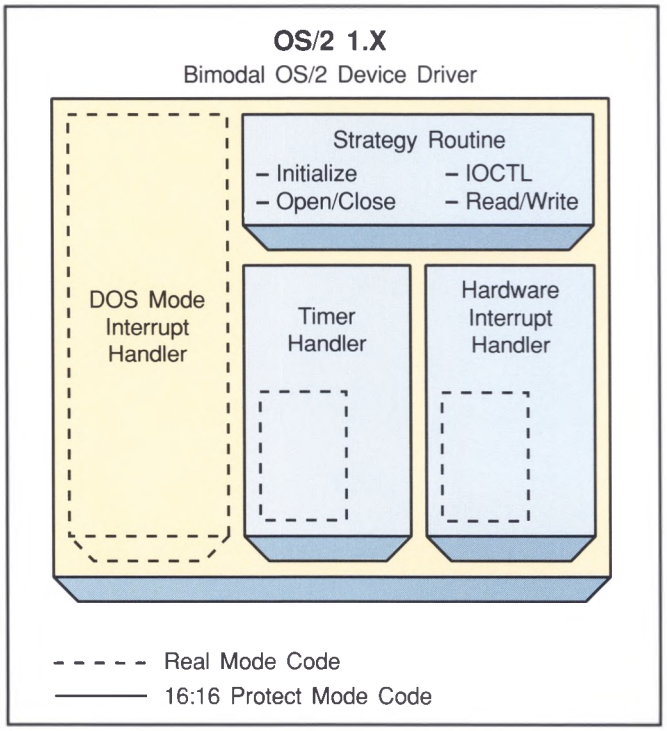


Figure 2. Structure of Bimodal Device Drivers

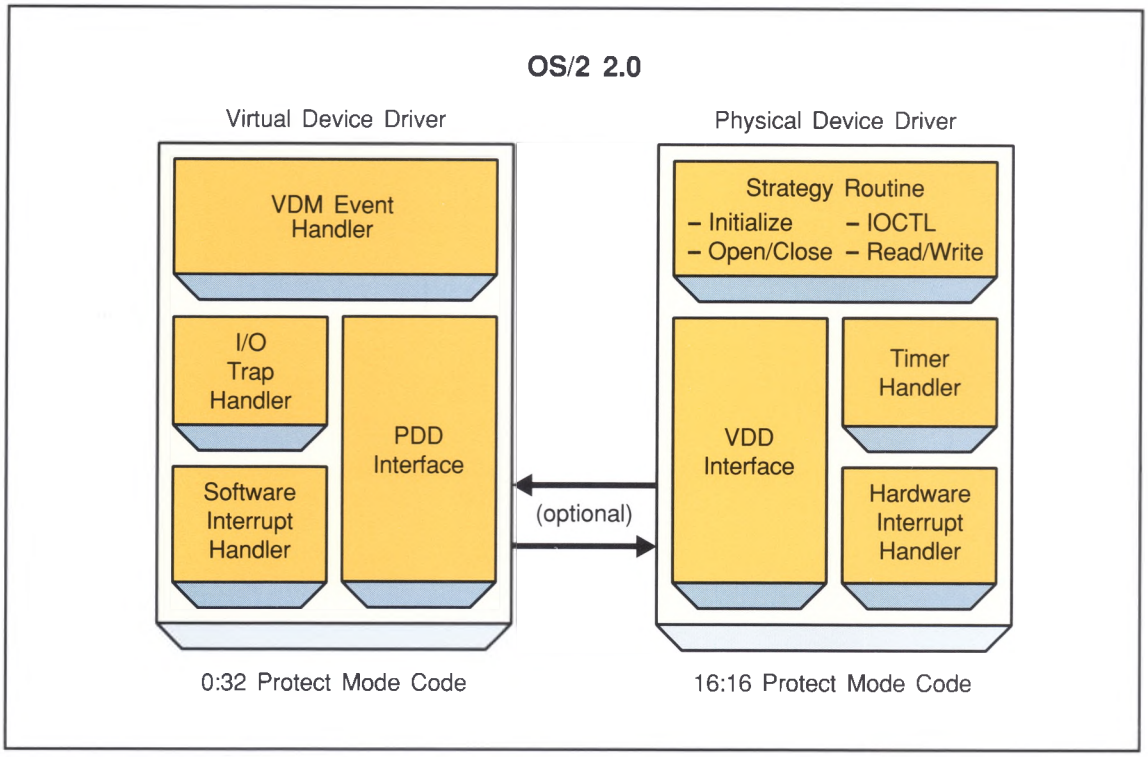


Figure 3. Structure of Virtual and Physical Device Drivers

VDD	Interfaces with PDD
BIOS	
CMOS/Real Time Clock	
Programmable Interrupt Controller	
Timer	X
Keyboard	X
Mouse	X
Disk/Diskette	X
DMA Controller	
Video (EGA, VGA, and 8514/A)	
Printer	X
COM	X
Expanded Memory Specification	
Extended Memory Specification	
Numeric Coprocessor (80387)	

Figure 4. Virtual Device Drivers

The Virtual Device Driver Model

All OS/2 1.X device drivers are considered bimodal, since they are required to run in either real mode or protected mode. This is true even if the device driver does not provide explicit DOS mode support (such as hooking a DOS mode software interrupt). Figure 2 shows the structure of bimodal OS/2 device drivers. Because MVDM eliminates real-mode execution, the existing device drivers were updated to remove the real-mode sections. These drivers are now referred to as Physical Device Drivers (PDDs).

Virtual Device Drivers are new in OS/2 2.0. VDDs are written as 32-bit, flat memory model, protected-mode modules. Figure 3 shows the structure of Virtual Device Drivers and the interfaces to a PDD. Nearly all VDDs provided in OS/2 2.0 are written in a high-level language (C), although several have portions that were developed using assembly language.

All VDDs are loaded and initialized during the system initialization (boot) process. Many of them establish communication at that time with the associated PDD. Figure 4 lists the VDDs provided in OS/2 2.0 and indicates those that have a direct interface with an OS/2 PDD. The VDD controls access by any DOS application to the device, and relies on the PDD to manage the physical hardware operations.

For example, the Virtual COM device driver has a direct interface to the Physical COM device driver. This interface allows the COM PDD to service all the hardware interrupts and to buffer data being transmitted or received. The Virtual COM driver emulates asynchronous communications BIOS functions (INT 14H) to send and receive characters and to set or query the state of a COM port. The Virtual COM driver also virtualizes the I/O ports associated with a COM port.

This allows the Physical COM driver to manage the actual COM port hardware, while a DOS application accesses only a virtual copy of the port. Even with this design, an OS/2 application may operate on one COM port while a DOS application accesses a second COM port. Both of these applications can perform data communications in the background. The relationship between the VDD and PDD is shown in Figure 5.

Not every VDD needs to operate with a PDD in the same manner as indicated for COM port virtualization, as shown from the list in Figure 4. The VDD-PDD interface is required when the device generates hardware interrupts that must be simulated in the context of the Virtual DOS Machine. For example, many DOS applications that support asynchronous communications include hardware interrupt handler routines. These routines typically perform I/O directly to the COM

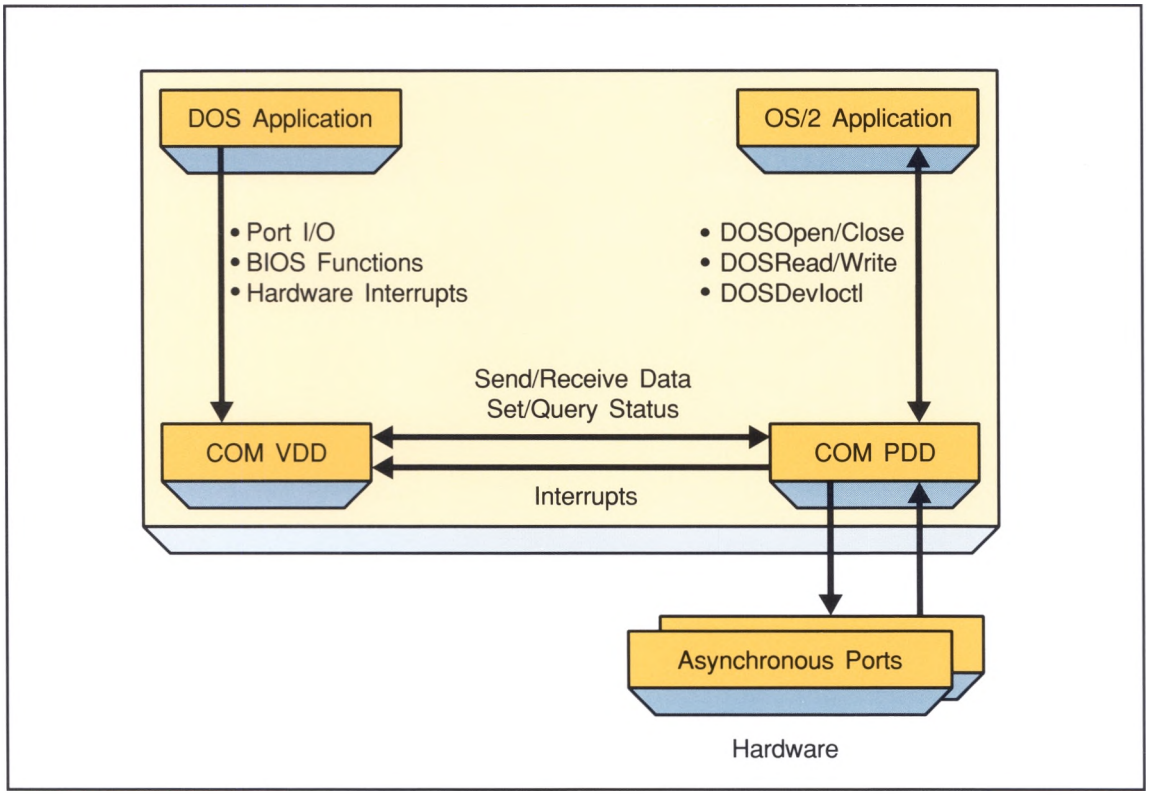


Figure 5. Virtual COM and Physical COM Device Drivers

OS/2 2.0 allows protected-mode OS/2 processes to request services directly from a VDD.

port hardware rather than going through BIOS.

To support these DOS applications and to allow them to run in the background and foreground, the Virtual COM driver simulates hardware interrupts in the task-time context of the V86 mode process. Virtual DOS Machines are scheduled to run using the same preemptive time-slicing, task-dispatching method that drives the multitasking OS/2 sessions. Hardware interrupts, on the other hand, occur asynchronously to this task-scheduling process. By simulating the hardware interrupts and presenting a virtual hardware state, the interrupt handling logic of the DOS application does not execute on the physical interrupt thread. This means that switching to V86 mode is not done at interrupt time, but deferred until the scheduler dispatches the VDM task.

The advantage of simulated interrupts is that mode switching and hardware virtualization do not need to be done at interrupt time. Also, the DOS application does not get control at interrupt time, which helps to maintain system integrity.

A potential disadvantage of this approach is that DOS applications with routines that are highly real-time dependent may not operate correctly under heavy system load conditions. When developing a VDD, it is important to consider the time-dependent aspects of the DOS application, and to design the hardware virtualization to satisfy this dependency. This can usually be achieved when all hardware states can be simulated in software, and where performance considerations remain high on the VDD-PDD designer's agenda.

Communication with OS/2 Processes

Another feature of OS/2 2.0 allows protected-mode OS/2 processes to request services directly from a VDD. New OS/2 Application Programming Interfaces (APIs) support this capability.

The VDD designer may provide any appropriate services across this interface, depending on the hardware being virtualized. Potential uses of this interface include the ability of OS/2 applications to communicate with DOS applications. To achieve this, the VDD would also have to provide DOS applications with an interface to establish the communications. This could be done, for example, by defining a series of software interrupt functions that a DOS application could invoke.

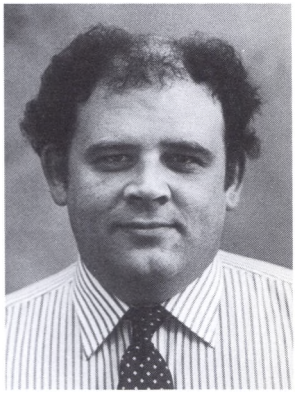
John G. Tyler, *Entry Systems Division, IBM Corporation, 1000 NW 51st Street, Boca Raton, FL 33429. Mr. Tyler joined IBM in Boca Raton in 1984 as a programmer for the Graphics Development Toolkit. In 1985, he became a team leader in OS/2 development, responsible for the asynchronous communications device driver. In 1988, Mr. Tyler became the lead designer for OS/2 DOS compatibility and MVDM. Mr. Tyler holds an MS in Computer and Information Sciences and an EdS in Educational Media and Instructional Design, both from the University of Florida.*

32-Bit OS/2

OS/2 Numeric Support

by Tony Ingenoso

Before the 80486, Intel 80X86 family CPUs required the use of an optional coprocessor chip to perform hardware-assisted, floating-point math. The 80486, with an integrated math unit, eliminates the need for a separate coprocessor chip. If there is no coprocessor in a system, software must deal with floating-point math via software emulation of the math coprocessor function. With the OS/2 system, an application can automatically use hardware math support, if present, and math emulation if not.



Tony Ingenoso

MATH OPERATIONS IN OS/2 1.X

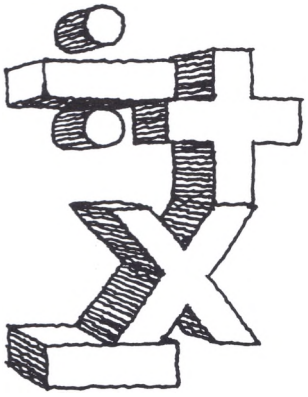
OS/2 has provided system-level support for 80287 and 80387 math coprocessor chips since its first release. In OS/2 versions 1.0, 1.1, and 1.2, each thread in the system thinks that it has exclusive use of the coprocessor, even when other threads are doing math operations.

In OS/2 1.X, the DosDevConfig Application Programming Interface (API) detects the presence of a coprocessor. This API does not identify which kind of coprocessor is present (80287, 80387, or the 80486's built-in math unit). The application must determine this by a process of elimination.

Although an 80287 uses both "projective" and "affine" infinities, the 80387's and the 80486's math units use only the affine mode; this is how to distinguish an 80287 from an 80387 or 80486. Figure 1 is a sample program that determines which of these is present. This program can run on all versions of OS/2, and is built using the following commands:

```
masm  npxtype;
link  npxtype,npxtype,,c:\os2\doscalls;
```

The presence of an 80386 CPU does not imply an 80387 math unit. Several non-IBM systems use an 80386/80287 combination, since Intel designed the 80386 to work with either the 80287 or 80387. This combination is crippled functionally, however, because the 80287 does not know how to run in the flat memory model introduced in OS/2 2.0. The 80287 only knows how to deal with segmented 80286 style addresses. All currently released 80386 PS/2's use only the fully functional 80386/80387 combination. If OS/2 2.0 detects an 80386/80287 combination, the 80287 is disabled and the 80387 emulator described below is used instead.



If no coprocessor is present, OS/2 1.X allows an application to install a math emulator of its own via the DosSetVec system API. A handler for the INT 7 vector (coprocessor not present) is installed, and an emulator operates via this handler. Nothing prevents an application from installing an emulator, even if a coprocessor is present under OS/2 1.X. For the typical high-level language programmer, this happens behind the scenes and is handled by the start-up code and math library for the language used.

OS/2 1.X deals with exceptions in a tricky way. These versions register handlers on a per-process rather than a per-thread basis. Emulators running under OS/2 1.X wishing to support multithreaded math must keep track of the floating-point contexts on a per-

32

If no coprocessor is present, OS/2 1.X allows an application to install a math emulator of its own via the DosSetVec system API.

```
STDOUT = 1
        .286p
        .287
        extrn   DOSSETVEC:FAR
        extrn   DOSDEVCONFIG:FAR
        extrn   DOSWRITE:FAR
        extrn   DOSEXIT:FAR

_stk    segment para stack 'stack'
status  dw      ?                       ; NPX status word
npxflag db      ?                       ; Flag from DosDevConfig()
        org     0
        db      4096 dup (?)            ; 4K stack
_stk    ends

dgroup  group   _stk

_cod    segment para public 'code'
        assume  cs:_cod,ss:_stk,ds:_stk
msg1    db      'The co-processor '
msg1len EQU     $-msg1
msg2    db      'is not present.',13,10
msg2len EQU     $-msg2
msg3    db      'is a 80287.',13,10
msg3len EQU     $-msg3
msg4    db      'is a 80387.',13,10
msg4len EQU     $-msg4
msg5    db      'is a 80486 math unit.',13,10
msg5len EQU     $-msg5

start   label   near
;-----------------------------------------------------------------------------
; DosDevConfig(&npxflag, 3, 0);
;-----------------------------------------------------------------------------
        push    ss
        push    offset npxflag
        push    3
        push    bp                      ; BP=0 on entry
        call    DOSDEVCONFIG
        cmp     npxflag, 0
        jne     got_something           ; something is there
        mov     bx, offset msg2         ; No math unit present
        mov     cx, msg2len
        jmp     terminate
;-----------------------------------------------------------------------------
; Test for projective infinity.
;-----------------------------------------------------------------------------
got_something:
        finit                   ; Set a default control word
        fld1                    ; Create
        fldz                    ;    an
        fdiv                    ;       infinity
        fld st                  ; Dup the infinity
        fchs                    ;    and change the sign
        fcompp                  ; Compare the 2 infinities
        fstsw   status          ; A 287 will say they are equal
        mov     ax, status      ;    a 387/486 will say they are
        sahf                    ;       different.
        jne     its387_486      ;
        mov     bx, offset msg3 ; A 80287 is present.
        mov     cx, msg3len
        jmp     terminate
```

Figure 1. Sample Program to Determine Which Coprocessor is Present (Continued)

```
;-----------------------------------------------------------------------
; When this handler is executed it means a 386 is the CPU.  Int 6 is the
; invalid opcode exception.
;-----------------------------------------------------------------------
int6trapper label near
      mov     bx, offset msg4 ; A 80386 is present.
      mov     cx, msg4len
      jmp     terminate
;-----------------------------------------------------------------------
; Determine if we are running a 386 or 486.  We register an Int 6 handler
; and then try to execute one of the 486 specific opcodes.
;-----------------------------------------------------------------------
its387_486:
      push    6                       ; Int 6
      push    cs                      ; &handler
      push    offset int6trapper      ;
      push    ss                      ; A handy dummy at SS:0
      push    bp                      ; bp==0
      call    DOSSETVEC               ;

      db      0fH, 0C0H, 0C0H         ; A 486 CMPXCHG instruction

      mov     bx, offset msg5 ; A 80486 is present.
      mov     cx, msg5len
      jmp     terminate
;-----------------------------------------------------------------------
; DosWrite(stdout, msg1, msg1len, &SS:0);
;-----------------------------------------------------------------------
terminate:
      push    STDOUT                  ; Handle
      push    cs                      ; &string
      push    offset msg1
      push    msg1len                 ; string len
      push    ss                      ; SS:0 is a handy for BytesWritten
      push    bp                      ; bp==0
      call    DOSWRITE
;-----------------------------------------------------------------------
; DosWrite(stdout, cs:bx, cx, &SS:0);   ; BX->string, CX==message length
;-----------------------------------------------------------------------
      push    STDOUT                  ; Handle
      push    cs                      ; &string
      push    bx                      ; cs:bx->string
      push    cx                      ; cx==string len
      push    ss                      ; SS:0 is a handy for BytesWritten
      push    bp                      ; bp==0
      call    DOSWRITE
;-----------------------------------------------------------------------
; DosExit(0, 0);
;-----------------------------------------------------------------------
      push    bp                      ; kill this thread
      push    bp                      ; RC==0
      call    DOSEXIT
_cod    ends
      end     start
```

Figure 1. Sample Program to Determine Which Coprocessor is Present

thread basis and look at the current thread ID when operating to know which context to use. Often language vendors avoid this complexity and support emulated floating-point operations for a single thread in a process. The IBM C/2 compiler version 1.0 is one of these. IBM C/2 version 1.1 supports multiple threads performing emulated floating-point operations.

A system-provided 80387 emulator is available to all user applications running in protected mode.

DOS MODE OPERATIONS UNDER OS/2 1.X

OS/2 fully supports DOS application usage of a coprocessor in the DOS-compatibility mode. A DOS application will appear to have exclusive use of the coprocessor, even though OS/2 protected-mode processes are using the same coprocessor in the background.

DOS applications perform emulation in two ways when no math coprocessor is present. In one scheme, the language compiler generates special software interrupts rather than actual 80287 or 80387 opcodes. If a coprocessor is present when one of these applications runs, the software interrupts are typically "back patched" with the corresponding 80287 or 80387 opcodes. Therefore, the coprocessor, not the emulator, is used the next time the instruction is hit. The other scheme is specific to CPUs that support the "coprocessor not present" exception (80286, 80386, and 80486). These applications install a handler for INT 7. They then set a bit in the Machine Status Word (MSW) of the processor to generate an INT 7 if an attempt is made to execute a coprocessor opcode. This style of DOS math emulator is architecturally similar to that used in protected-mode OS/2 applications.

The 1.X versions of OS/2 support operating both styles of floating-point emulation. The language offerings from Microsoft and Borland include some typical implementations of the software interrupt-based emulators under DOS. A program called EMUL87, a DOS terminate-and-stay-resident (TSR) program available on many Bulletin Board Systems (BBSs), is an example of the INT 7-based emulation technique.

OPERATION UNDER OS/2 2.0 PROTECTED MODE AND V86 MODE

There are significant changes in the way OS/2 2.0 supports numeric operations. A system-provided 80387 emulator is available to all user applications running in protected mode. This means that all applications intended for OS/2 2.0 can be compiled safely with options that generate inline X87 code. Assembler programmers can code as if a coprocessor is always present. This emulator is available to both native 32-bit applications and to older 16-bit applications. The system emulator does not, however, support DOS applications running in Virtual 8086 (V86) mode. DOS applications must continue to use whatever mechanisms they are currently using for floating-point emulation.

In OS/2 2.0, when the DosDevConfig API asks if a coprocessor is present, the application is always told that it is. In fact, if a 16-bit application tries to install its own emulator with DosSetVec, that call will fail, and the application will still use the system-provided 80387 emulator. A compatibility problem may occur. If a 16-bit application really needs to do math in the projective infinity mode[1], and wants to install its own 80287 style emulator because it noticed that it is running on an 80387 (which supports only affine-infinity mode), this application may return

[1] The IEEE 754 draft standard for binary floating-point arithmetic proposed two modes of processing for "infinity" values: the "protective" mode (where an infinity is a signless quantity), and the "affine" mode (where there can be positive and negative infinities). The draft also specified the projective mode as the default. The 8087 and 80287 coprocessors were based on draft versions of the IEEE 754 document. When IEEE 754 finally became solid, the notion of the projective infinity had been dropped. Thus, the 80387 and the 80486 math unit, which were designed after the standard became firm, do not operate in projective mode. To maintain compatibility with older code, these chips allow the bit in the math unit's control word that specifies projective infinity to be flipped. They continue to operate in the affine mode. Serious compatibility problems may occur when older code is run on one of the affine-only processors if the older code depends specifically on the behavior of a projective infinity mode of operation. Most of the time this should not be a problem, but the potential does exist.

incorrect results or act in an unpredictable way because its DosSetVec call to install its emulator failed. It will run instead on the affine-only system emulator. The chances of this being a problem for compiler generated code are slight. Assembler programmers, however, should keep this in mind.

Users, who actually have an 80387 chip in their systems may have the chip disabled by OS/2 2.0 during boot, which causes the system emulator, not the chip itself, to be used.

REFERENCES

IBM. *OS/2 Technical Reference, Version 1.0.* 1987.

IBM. *OS/2 Technical Reference, Version 1.1.* 1988.

IBM. *OS/2 Control Program Programming Reference, Version 1.2.* 1989.

Intel. *The 8086 Family User's Manual, Numerics Supplement.* 1980.

Intel. *80286 Programmer's Reference Manual, Numerics Supplement.* 1985.

Intel. *80387 Programmer's Reference Manual.* 1987.

Intel. *i486 Microprocessor Data Sheet.* 1989.

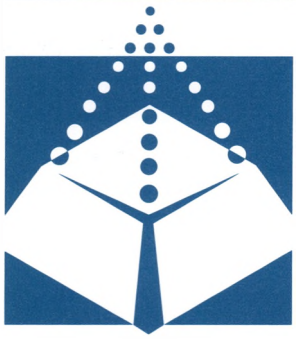
Coonen, J.; Kahan, W.; Palmer, J.; Pittman, T.; and Stevenson, D. "A Proposed Standard for Floating-Point Arithmetic," *ACM SIGNUM Newsletter* (October 1979).

Coonen, J. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *IEEE Computer* 13 (January 1980).

IEEE Task P754. "A Proposed Standard for Binary Floating-Point Arithmetic," *IEEE Computer* (March 1981).

Tony Ingenoso, *IBM Entry Systems Division, 1000 NW 51st Street, Boca Raton, FL 33429.*
Mr. Ingenoso joined IBM in 1989, and is currently a staff programmer in OS/2 2.0 System Test. He previously held positions with Borland International (Turbo C development team) and Rockwell International's Space Division (Space Shuttle flight software). He holds a BA from SUNY Potsdam and an MS in Computer Science from the University of California at San Diego.

DOS applications must continue to use whatever mechanisms they are currently using for floating-point emulation.



OS/2 Standard Edition

Performance Tips and Guidelines

Keith Culley

Barry Diller

by Keith Culley, Barry Diller, Terry Green, and Roger Lewis

A period of uncertainty for programmers always follows the arrival of a new operating system such as OS/2. Initially, there are few known rules for optimizing program performance. Programmers gain such knowledge from trial-and-error experimentation and from performance analysis of completed programs. This article contains a set of tips and guidelines to help programmers write OS/2 programs with optimum performance.

The information in this article was compiled from published OS/2 books, technical journals, discussions with dozens of OS/2 programmers and analysts within IBM (representing many hours of OS/2 programming experience), and drawn from our knowledge of the operating system. The information is divided into the categories of memory management, packaging, and program execution control.

MEMORY MANAGEMENT

Memory management refers to the function of the OS/2 operating system that allocates available memory to the concurrent multitasking processes that are competing for the physical and virtual memory. Memory, beyond that which is needed by the OS/2 modules and data structures, is administered by the OS/2 memory manager, which allocates and releases segments as required by the applications.

Use of Segments

The standard unit of memory allocated by all OS/2 versions through 1.2 is called the *data segment*, which can be up to 64 KB. Segment *descriptors* contain the attributes and location of physical memory of each segment. These are stored in descriptor tables maintained by OS/2. *Selectors* are used as indexes (or handles) into these tables. Each OS/2 process has its own Local Descriptor Table (LDT), which contains references to all segments associated with a process. A Global Descriptor Table (GDT) contains references to segments that are available to all processes in the system.

At program load time, the OS/2 loader creates an LDT for the program and makes an entry into the table for each segment defined in the program. If a segment entered into the LDT is defined as "Preload," that segment is loaded into memory at program load time. If a segment is defined as "Load-on-call," the descriptor for that segment is marked "not present," and the segment is not loaded. When, or if, the segment is eventually referenced, the "not present" indication causes OS/2 to load the absent segment from disk.

Whenever a segment is entered into an LDT, OS/2 must reserve memory for it. If physical memory necessary to satisfy a segment load request is not available, OS/2 first attempts to combine fragmented free memory areas. If this compaction fails to recover enough memory to load the new segment, OS/2 then employs a "least-used" algorithm to determine which currently loaded segments must be swapped to disk. It examines an internal time-stamp value for each segment in the system to identify the earliest referenced segments. These segments are then moved to the system swap file. The descriptors associated with the swapped segments are marked "not present," and another attempt is made to load the new segment. If swapping fails to recover enough free memory for the new segment to be loaded, compaction is tried again.

Grouping of Functions

An application's performance can be greatly affected by the organization of code and data within program segments. Careful selection and segregation of code and data segments can, in some cases, result in an order-of-magnitude improvement in performance over more *laissez-faire* organization strategies. There are no rigid rules for placing code and data within program segments, but there are some basic guidelines:

- Group all "mainline" program code into one segment. This includes the program main procedure and any functions frequently referenced by the main procedure. This segment can be large, approaching the 64 KB limit, since it is used frequently and will be swapped out rarely, if ever.

- Place error processing code and application initialization code in separate segments. If these pieces of code are small, combine them into a single segment. Other infrequently used code should also be combined into a single segment. Placing rarely used code in separate segments provides OS/2 with some obvious candidates for swapping, if necessary.

- Put hardware-dependent code into individual segments. The code for particular hardware devices should be placed in separate segments unless the devices are commonly used together. This allows OS/2 to avoid loading unnecessary device-support code within a segment when loading or swapping.

- Place functions that call each other frequently within the same code segment. If functions are on separate segments, and the segments are not in memory at the same time, swapping is required, thereby requiring expensive CPU time.

- Keep the total number of code and data segments small. A few large segments are better than many small ones, because small segments can cause additional, unnecessary work for the OS/2 memory compaction and swapping functions (swapping small segments out of memory requires almost as many resources as swapping large segments, but releases less memory).

Compiling and Linking

When a C source file is compiled into an .OBJ file (or module), the compiler, by default, places the program instructions for all C functions into one or more (depending on the memory model) "text segments." (This is the terminology used in the compiler reference manual. It refers only to a block of code, not an OS/2 segment.) Then the OS/2 linker, by default, tries to pack all text segments from all .OBJ files into a single OS/2 segment. The /PACKCODE option can be used to force the linker to pack text segments only up to a specified number of bytes before using another OS/2 segment. This allows the programmer to set an upper limit on the size of segments created.

To follow the segment grouping guidelines in "Grouping of Functions" above, the programmer must force the OS/2 linker to group specific compiled text segments and to place each group within its own OS/2 segment. There are different ways this can be done, as shown below.

1. Examine the size of all text segments, carefully order their input to the linker, and use the /PACKCODE option to force adjacent functions into common OS/2 segments. This is an iterative process and is unsuitable for large applications since it may require creating "dummy" functions to force the segment breaks.

2. Use the /NT compiler option to name all text segments explicitly and to give identical names to the text segments that are to be grouped into the same segments. Then, link the segments, using either the /NOPACKCODE linker option or the SEGMENTS parameter in the linker DEF file. The linker groups all text segments with identical names and puts each text segment into its own OS/2 segment. This is the recommended method.

Figure 1 is an example MAKE file for program NPP. The goal is to place functions from the files NPPINPUT.C and NPPDIAL.C into a common OS/2 segment. To do this, both files are compiled with the /NT option; identical text segment names (INPUT_TXT) are then assigned to the compiled code for these files.

Terry Green

Roger Lewis

```
npp.exe:  npp.obj nppinput.obj nppdial.obj nppcomm.obj rpterr.obj npp.def npp.res
  link @npp.l
  rc $*.res

npp.obj:      npp.c npp.h nppdef.h
  cl /c /Alfu /W2 /G2s /Zlp /NT MAIN_TEXT $*.c
nppinput.obj: nppinput.c npp.h nppdef.h
  cl /c /Alfu /W2 /G2s /Zlp /NT INPUT_TEXT $*.c
nppdial.obj:  nppdial.c npp.h nppdef.h
  cl /c /Alfu /W2 /G2s /Zlp /NT INPUT_TEXT $*.c
nppcomm.obj:  nppcomm.c npp.h nppdef.h
  cl /c /Alfu /W2 /G2s /Zlp /NT COMM_TEXT $*.c
rpterr.obj:   rpterr.c
  cl /c /Alfu /W2 /G2s /Zlp /NT _TEXT $*.c
npp.res:      npp.rc npp.dlg nppdef.h npp.ico

  rc -r $*.rc
```

Figure 1. Segment Packing Example MAKE File

The compiled code is then linked with the /NOPACKCODE (abbreviated /NOP) switch, as shown in Figure 2.

```
npp.obj +
nppcomm.obj +
nppinput.obj +
nppdial.obj +
rpterr.obj      /A:16
npp.exe
npp.map         /NOD /MAP /NOP
llibce.lib+
os2.lib
npp.def
```

Figure 2. Segment Packing Example Link File

The resulting MAP file in Figure 3 shows the following:

- The code for NPP (text segment MAIN_TEXT) was placed in a single OS/2 segment 0001.
- The code for NPPCOMM (text segment COMM_TEXT) was placed in segment 0002.
- The code for both NPPINPUT and NPPDIAL (text segment INPUT_TEXT) was placed in a common segment 0003.
- The code for RPTERR (text segment _TEXT) was placed in segment 0004.

Refer to the manual for the C/2 Compile, Link, and Run, Version 1.1 for help in interpreting linker MAP file output.

Segment Loading

All segments, whether data or code, should be specified as Load-on-call, not Preload. The system should load segments only when actually referenced rather than at program load time. By default, the linker makes all code and data segments Load-on-call. This setting, however, may be overridden by using the Code, Data, and/or Segments statements within the linker module definition (.DEF) file.

MEMORY ALLOCATION

Suballocation

Frequent allocating and freeing of segments and objects may result in poor system performance. One solution is to allocate blocks (small sections of memory within an existing segment) via the OS/2 DosSubSet and DosSubAlloc functions. Allocation of many small segments (4 KB or less) via DosAllocSeg should especially be avoided, since it can cause extra work for the OS/2 memory compaction and segment-swapping functions. A better solution is to use small blocks suballocated from a single segment via DosSubAlloc. This allows a program to access all blocks without requiring a reload of the processor-segment register for each (which would be time consuming).

There are caveats associated with the use of the blocks suballocated from segments, for example, the lack of memory protection between blocks.

Discardable Segments

Segments allocated with DosAllocSeg or DosAllocHuge can be made discardable by setting the appropriate flag parameter. OS/2 can automatically overwrite flagged discardable segments if the memory space occupied is needed, thereby enhancing performance. This is appropriate when the application needs to store data briefly, or when the data can be quickly regenerated. DosLockSeg and DosUnLockSeg lock and unlock discardable memory segments.

Memory Deallocation (Clean-Up)

Memory allocated by an application should be freed as soon as the application no longer needs it. Freeing unused memory reduces the need for OS/2 to swap segments in a loaded system, thus providing more useful CPU time for running applications.

Allocated Segments and Blocks

DosFreeSeg frees memory allocated via DosAllocSeg, DosAllocHuge, and DosAllocShrSeg when no longer needed. DosSubFree frees memory blocks suballocated within a segment via DosSubSet and DosSubAlloc. These should be executed when the memory allocated is no longer needed.

Shared Memory

OS/2 maintains a usage count for shared segments. Each time a process accesses a segment via the DosGetShrSeg call, or gives another process access to a segment it owns via the DosGiveSeg call, the usage count for the segment is incremented. Each use of DosFreeSeg against the segment decrements the usage count. A segment is freed only when DosFreeSeg decrements the segment usage count to zero.

Thread Stack Space

Memory allocated to serve as stack space for a created thread is not automatically freed when the thread terminates. An application that repeatedly allocates OS/2 segments as stack space for new threads, but never frees those segments, may eventually tie up many selectors and associated memory.

Be cautious when freeing stack space used by threads. A thread that is terminating must inform some other thread, via semaphore, that it is exiting and will no longer need the allocated stack space. The thread must also execute DosExit before the other thread frees the stack space. If the "clean-up" thread kills the terminating thread's stack before DosExit is executed, the call will fail and the application will suffer a memory-protection violation error.

```
.
 NPP

Start         Length      Name          Class
0001:0000     01830H      MAIN_TEXT     CODE
0002:0000     0019AH      COMM_TEXT     CODE
0003:0000     00638H      INPUT_TEXT    CODE
0004:0000     00734H      _TEXT         CODE
0005:0000     00000H      C_ETEXT       ENDCODE
0005:0000     0000CH      FAR_BSS       FAR_BSS
0006:0000     00042H      NULL          BEGDATA
0006:0042     00574H      _DATA         DATA
0006:05B6     0000EH      CDATA         DATA
0006:05C4     00000H      XIFB          DATA
 .     .      .           .             .
 .     .      .           .             .
 .     .      .           .             .
```

Figure 3. Segment Packing Example MAP File (Partial)

The following procedure is one method to prevent freeing a thread's stack memory before it can terminate:

1. A "creator" thread executes a DosAllocSeg to acquire stack memory for a new thread, and then creates the thread via DosCreateThread.

2. Some "clean-up" thread executes DosSemSetWait and waits for the new thread to clear the semaphore.

3. The newly created thread does its assigned work, and then it terminates as follows:

   - Execute DosEnterCritSec to freeze the "clean-up" thread.

   - Execute DosSemClear to indicate to the "clean-up" thread that stack space can be freed.

- Execute DosExit, which also performs an implicit DosExitCritSec.

4. The now unblocked "clean-up" thread frees the terminated thread's stack memory via DosFreeSeg.

Loaded DLLs

A program's links to a Dynamic Link Library (DLL) module are discarded when the program terminates. As a rule, however, DosFreeModule should be used to free a DLL as soon as the program is finished with it. Otherwise, the module remains in memory until all programs accessing it either terminate or execute DosFreeModule (except for normal segment swapping). In other words, if OS/2 does not know the program is finished with a DLL, it continues to swap it as usual, thus wasting system resources.

Resource Segments

Resources allocated via the DosGetResource call must be freed via the DosFreeSeg call. DosFreeSeg must be executed against a resource segment as many times as DosGetResource was executed within the process. If the resource segment was defined as Load-on-call, the segment is then freed. If the resource segment was defined as Preload, an additional DosFreeSeg call against the segment must be made. The additional call is required because Preload carries an implicit DosGetResource, executed at application load time.

The performance of an OS/2 application can be affected by its placement within the machine's memory.

MEMORY PERFORMANCE

The performance of an application running under OS/2 on a PS/2 can be affected by its placement within the machine's memory. PS/2 memory is divided into two areas: faster memory, added to the PS/2 planar board main bus, and slower memory, added to cards plugged into the Micro Channel bus. System code, the DOS box, and device drivers are all placed into the lower, faster memory. OS/2 applications are loaded into the slower, high-end memory. Program code placed in memory on the planar board runs much faster than the same program code placed in high memory. Thus, application performance may vary from run to run depending on the order in which applications are started. It may even vary within a single run if OS/2 compaction and swapping operations occur.

PACKAGING

Packaging refers to the assembly of source code into an optimal arrangement; in this case, it refers to the way the various individual elements are compiled and linked into the final executable application and its associated DLLs.

Figure 4 summarizes the overall flow of the compile and link processes when generating an application.

The rational used in packaging may have various motives, performance, marketing, or maintenance. Marketing considerations may require certain functions to be separate because these capabilities are being sold separately, even though, from a performance viewpoint, the code may belong together. Similarly, maintenance requirements may require common function grouping to more easily track and isolate issues. This article examines packaging from a performance viewpoint.

APPLICATION LINKING METHODS

Linking methods are important because they affect the level of performance of an application. An understanding of the basic linking methods, static and dynamic, will enable developers to obtain the optimum organization of functions and segments within executable modules.

With static linking, all application functions are resolved during the compile/link process. The final executable module is a complete, independent, stand-alone unit. A primary disadvantage of static linking is that common functions are not shared during execution. Each application has its own copy in memory.

Dynamic linking is different in that numerous programs may share common functions. Routines accessed by an application are resolved either when the application is loaded into memory (load-time linking) or when the application is ready to access the function during execution (runtime linking). These terms are not the same as Load-on-call

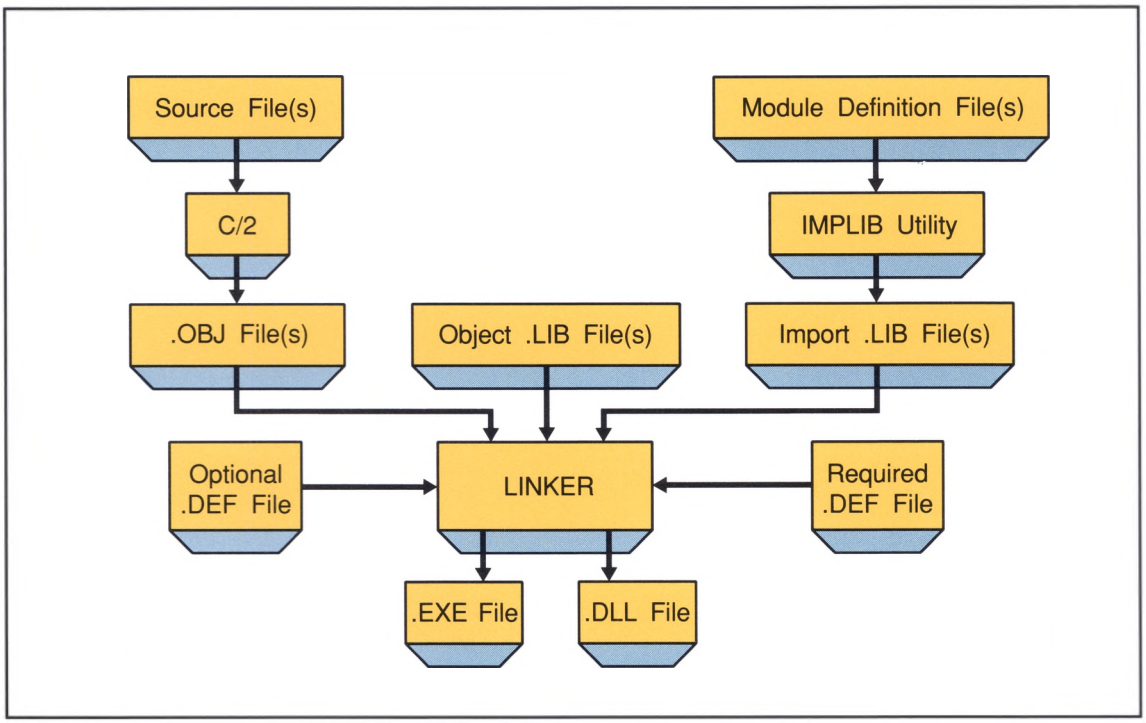


Figure 4. Overall Compile and Link Process

or Preload, which are types of loading (although related to linking, they are not types of linking). *Linking* refers to the process of resolving references to a function; *loading* refers to actual physical placement of the function into computer memory.

Static Linking

Static linking is best used when a program uses a relatively small function frequently, or when response time is critical. The drawback is that other programs using the same function also have their own copy of the object code in their .EXE file, thus increasing the overall memory requirements.

Object files to be linked can be combined into a single library by using the C/2 Compiler LIB command. This creates a library file containing the object modules, known as an *object library*. The developer then only needs to specify the .LIB object library, as opposed to specifying each .OBJ module individually.

Figure 5 links sales.obj, inventory.obj, and revenue.obj into one executable module. Figure 6 demonstrates how to combine these files into the ACNT.LIB library, using the LIB command.

Object libraries are often confused with the import libraries used in dynamic linking because both end with the .LIB file extension. Import libraries are created with the IMPLIB command. They contain header information about object modules, but not the modules themselves. Both import and object libraries are specified as parameters of the LINK command in exactly the same format. In fact, a combination of import libraries and object libraries can be included in the LINK command.

FUNCTION	DESCRIPTION
:MAIN.C	Sample program name of main procedure.
:MAIN.OBJ	Compilation of main procedure.
:SALES.OBJ	Function to compute daily sales. OBJ was created by compiling sales.c source code.
:INVENTORY.OBJ	Function to compute inventory. OBJ was created by compiling inventory.c source code.
:REVENUE.OBJ	Function to compute daily revenue. OBJ was created by compiling revenue.c source code.

```
LINK MAIN+SALES+INVENTORY+REVENUE,MAIN;
```

Figure 5. Static Linking with Object Modules (.OBJ Files)

```
LIB ACNT+SALES;
LIB ACNT+INVENTORY;
LIB ACNT+REVENUE;

  The link step parameters now change as
  shown:

LINK MAIN,MAIN,ACNT;
```

Figure 6. Static Linking with Object Libraries (.LIB Files)

Dynamic Linking

With dynamic linking, called functions do *not* have to be included as part of the executable file (.EXE), but can be included in DLLs. These contain code, data, and resource segments like executable files. Many applications may share the same function loaded in memory. DLLs and EXEs can be changed independently, thus reducing maintenance costs.

As mentioned previously, there are two types of dynamic linking: load-time and runtime. With load-time linking, calls to functions contained in DLLs are resolved when the application is loaded into memory. With runtime linking, the actual link occurs during the application's execution.

Load-Time Linking: As the name implies, external links to functions are resolved when the invoking application is loaded into memory (the application .EXE file contains a table of all referenced DLLs). Header information of a referenced DLL specifies which segments inside the DLL are to be loaded with the application. If the Preload linker option was used in creating a function, then the segment in which the function resides is loaded into memory when the invoking application is loaded. If the Preload option created all the segments in the DLL, then the entire DLL is loaded (a negative memory impact).

If the Load-on-call linker option created the segment, then only the module header information is loaded with the application. From a performance standpoint, the Load-on-call linker option is preferable in most situations. This method of loading DLLs also applies to runtime linking.

With load-time linking, a user links a function contained in a DLL by specifying the function(s) in the IMPORT section of the .DEF file, or by using import libraries. From the previous static link example in Figure 6, if the functions SALES, INVENTORY, and REVENUE were stored in the dynamic link library ACNT.DLL, the linker (.DEF) file would contain the code fragment shown in Figure 7, a simple example of load-time dynamic linking of functions.

Runtime Linking: With runtime dynamic linking, the application (using the DosLoadModule function) links to the DLL when the application is ready to use the function. Runtime linking offers two advantages. First, the application can continue to execute even if the link to the routine cannot be established. In load-time linking, the application is terminated by the operating system if a module is not found. Second, runtime linking enables an application to handle National Language Support (the switching of text information into different languages while it is running, by separating the text-data modules). Figure 8 shows an example of runtime linking.

```
DESCRIPTION   'SAMPLE BUSINESS PROGRAM'
PROTMODE
IMPORTS       ACNT.SALES
              ACNT.INVENTORY
              ACNT.REVENUE
```

Figure 7. Dynamic Linking at Load-Time

Ordinals

Applications can be linked in two ways: an import library may match the libraries that are going to be used at runtime, or the names of the module and entry-point routines may be specified with the IMPORT statements in the module .DEF file. In either case, the linker builds the names of the libraries and routines into tables in the .EXE file header, with pointers to each reference to the libraries in the executable code.

Imported library routines can also be specified in a .DEF file by an Ordinal instead of a name. The Ordinal is the index to the routine's position in the library's entry-point table. Using this technique speeds dynamic linking and reduces the size of the executable

file, but may cause maintenance problems, particularly when dealing with commercial applications scattered at various customer locations. Ordinals cannot be changed from one version of the library to the next; that is, from one release of the application to the next. If the application no longer requires a particular Ordinal, "stub" code must be written to maintain artificially the existence of that Ordinal definition.

Summary of OS/2 Linking Methods

Figure 9 compares features among static linking, dynamic load-time linking, and dynamic runtime linking.

PACKAGING FUNCTIONS INTO DLLs

The following general guidelines should be considered when grouping functions into DLLs:

- It is more desirable to have fewer DLLs and larger segments than many DLLs and smaller segments. The general rule of thumb is "the fewer DLLs and larger memory segments, the better."
  - Inter-DLL function calls have a high overhead on their initial invocation (for example, table lookup, disk read), unless both function segments previously reside in memory. After the first call, their cost to the system is equivalent to an intersegment call.
  - It is best to place frequently executed code into as few segments as possible, and to group within one DLL.
- External APIs should be grouped into their own DLL.
- Hardware and device-dependent functions should be included in their own DLL.
- Rarely executed code segments, such as error codes, may be grouped into their own DLL if desired.
- Grouping DLLs by application function can be beneficial from a marketing and maintenance point of view, but the number of intersegment calls required should be monitored. From a performance point of view, intersegment calls should be limited as much as possible.

After dividing the application into a set group of DLLs, the next step is to optimize the placement of functions/routines into segments within the DLL to reduce intersegment memory calls (refer to "Grouping of Functions" above).

PROGRAM EXECUTION CONTROL

This section discusses the program management facilities that OS/2 makes available to the programmer, such as the APIs that allow one program to control another activity, or to synchronize all the concurrent activities.

```
rc = call DosLoadModule((STRL)objNameBuf,(USHORT)
ObjNameBufLength,'ACNT.DLL', ModuleHandle);
```

Note: To runtime link the function REVENUE contained in ACNT.DLL, use DosLoadModule. Calls to REVENUE can then be executed once the DLL is linked.

Figure 8. Dynamic Linking at Runtime

Processes and Threads

An application consists of a series of instructions. As the CPU passes through these instructions, it creates a *thread* of execution. A *process* is a collection of one or more threads that owns resources, such as memory, open files, devices, or connections to DLLs. All threads under a process share its resources. If an application needs to exchange data quickly, and does not need to be concerned with the management of resources, then starting an additional thread may be appropriate. If the application is concerned about resource management, a process would be a better alternative. A comparison of processes and threads is listed in Figure 10.

OS/2 is particularly useful for "spinning-off" units of work, which ordinarily would tie up the screen and prevent the user from performing other functions of the application. Generally, if the time required is longer than one-half second to respond to the user, the unit of work should be done in a background thread or process.

Capability	Static Linking	Load-Time Dynamic Linking	Runtime Dynamic Linking
Uses Object files (compiled source).	X	X	X
Routines contained in object files are linked prior to load and execution time, and are included as part of the application executable file.	X		
Each program has its own copy of functions linked in memory at all times while the program is resident (a potential memory problem).		X	
Linking is performed at execution/load-time for inclusion of DLLs.		X	
Linking is performed during runtime for inclusion of DLLs as needed by the program.			X
Loading external DLL routines accessed by an application at application invocation is dependent upon their DLL segment being marked Load-on-call or Preload (a linker option).		X	X
Only one copy of a DLL is resident in memory for use by multiple programs (solves potential memory problem).		X	X
Linking is a manual step required prior to execution each time a function or library used by the program changes.	X		
The program does not have to be relinked prior to execution if DLLs are changed.		X	X
The source program must be able to resolve all references at load time (must know name of DLLs being used, versus being passed a DLL name during runtime and linking during runtime).		X	
The source program does not resolve references to DLLs until DLL is needed during runtime. DLL names can be passed by other programs during runtime and runtime linking is performed.			X

Figure 9. Comparison of Linking Methods

Interprocess Communications

Processes communicate primarily through shared memory, pipes, and queues. These are not normally used for communications between threads since global data within a process is an efficient means of interthread communications.

Shared Memory: Shared memory, the simplest and fastest way to transfer data, is accomplished by a process allocating a "shared segment." Then, another process can access the segment and exchange information. Shared memory has no built-in synchronization; therefore, activities must be coordinated to preserve the data integrity (often using semaphores).

PROCESSES	THREADS
Program-sized	Function-sized
More overhead to create	Fast creation
Data sharing is slightly slower since the operating system is controlling resources.	Data sharing and using RAM semaphores is a little faster.
Owns resources, files, and memory.	Owns stack space and registers. Developer must synchronize resources access.

Figure 10. Processes vs. Threads Comparison

Pipes: Pipes are used for redirecting data from standard input and output devices. Pipes permit a process, rather than the programmer, to redirect data going to another process. Information must be copied into a pipe; therefore, pipes are slower than shared memory.

Queues: Queues are the most flexible means of interprocess communication. Data can be read in as First-In-First-Out (FIFO), Last-In-First-Out (LIFO), or according to priority of the message. Queues are normally used when one process reads data from several other processes. Because data is shared by passing an address rather than actual data, the queue method is very fast. A queue is preferred when large amounts of data must be transferred quickly.

PERFORMANCE GUIDELINES SUMMARY

Figure 11 summarizes numerous performance items to consider during application development. In applying this data, however, remember that there are exceptions to any rule.

REFERENCES

Cortesi, David E. "Dynamic Linking in OS/2." *Dr. Dobb's Journal* (December 1987) 18–28.

Duncan, Ray. *Advanced OS/2 Programming*. Redmond, WA: Microsoft Press, 1989.

Heller, Vic. "OS/2 Virtual Memory Management." *BYTE* (April 1988): 227–233.

Iacobucci, Ed. *OS/2 Programmer's Guide*. Berkeley, CA: Osborne McGraw Hill, 1988, G325-0402.

IBM Corporation. *IBM Personal Systems Developer*. (November 1988 and February 1989).

Lafore, Robert, and Norton, Peter. *Peter Norton's Inside OS/2*. New York: Brady Books, 1988.

Letwin, Gordon. *Inside OS/2*. Redmond, WA: Microsoft Press, 1988.

PC Magazine, 1988–1989, inclusive.

Petzold, Charles. *Programming the OS/2 Presentation Manager*. Redmond, WA: Microsoft Press, 1989.

The OS/2 Programmers Reference, Volumes 1,2,3. Redmond, WA: Microsoft Press, 1989.

Keith Culley, *IBM Systems Integration Division, 3700 Bay Area Boulevard, Houston, TX 77058. Mr. Culley is a staff engineer in the Space Station DMS Architecture department conducting performance analyses of a proposed fiber-optic, UNIX-compatible, 80386-based system. He has been with IBM since 1978, working primarily with space shuttle software verification. An assignment to the Commercial Systems group involved various OS/2 tasks, including the work reported here. He received a BSc with honors in Mechanical Engineering from the University of Southampton, England.*

Barry Diller, *IBM Industrial Sector Division, 3700 Bay Area Boulevard, Houston, TX 77058. Mr. Diller is a staff engineer in the Process Systems Architecture department of the Houston Process Industry Solution Center. He joined IBM in 1984 and has done commercial systems integration work, specializing in OS/2 since 1986. Mr. Diller holds a BS and an MS in Electrical Engineering from Oklahoma State University.*

Memory Allocation Considerations

- Suballocate memory within a segment to reduce the number of segments.
- Use discardable segments, which are written over when they are no longer needed.
- Free allocated memory that is no longer needed.
- Free thread stack space that is no longer needed.
- Have each program referencing a DLL module free it when the module is no longer needed.

Memory Management Considerations

- Have a few large segments rather than many small segments.
- Group mainline code into one segment (or as few segments as possible), including functions frequently called by the main program.
- Group infrequently used code into one segment.
- Place functions that call each other frequently in the same segment.
- Make segments Load-on-call rather than Preload.

Dynamic Link Library Considerations

- Establish separate DLL and .EXE code so they may be updated and recompiled separately.
- Use a DLL for shared functions. One copy of the DLL code is then loaded into memory and referenced by all programs using it.
- Use a separate DLL for hardware/device-dependent routines.
- Use a separate DLL for external APIs.
- Use Runtime, rather than Load-time, linking of a DLL.
- Use fewer and larger DLLs.
- Remember, using Ordinals can introduce maintenance difficulties.

Other Considerations

- Fill up planar board first since memory on the planar board (mother board) is faster than memory on adapter cards.
- Put the path of infrequently used DLLs at the end of the LIBPATH statement, thereby making directory search more efficient.

Figure 11. Performance Guidelines Summary

Terry Green, *IBM Industrial Sector Division, 3700 Bay Area Boulevard, Houston, TX 77058. Mr. Green is an advisory engineer in the Workstation System Development department of the Houston Process Industry Solution Center. He has been with IBM since 1967 and has done commercial systems integration work, specializing in OS/2, for the past year. Mr. Green holds a BS in Electrical Engineering from Louisiana State University and an MS in Engineering (Feedback Control Systems Specialization) from the University of Alabama in Huntsville.*

Roger Lewis, *Industrial Sector Division, 3700 Bay Area Boulevard, Houston, TX 77058. Mr. Lewis is a staff engineer assigned to the Advance Workstation Engineering department working in Process Industry Solutions. He joined IBM in 1983 working on space shuttle simulator development and support. This was followed by an assignment in the Commercial Systems Integration area where he specialized in OS/2-based solutions. Mr. Lewis holds a BS in Computer Science from the University of Southwestern Louisiana.*

OS/2 Standard Edition

Writing OS/2 Device Drivers in C

Raymond Westwater

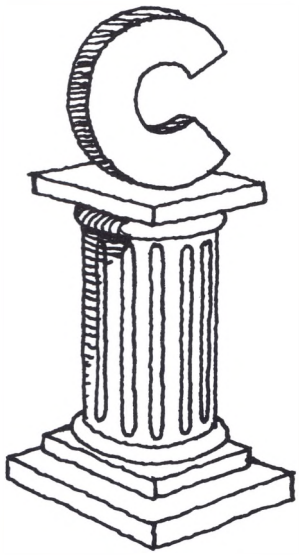
by Raymond Westwater

Although no C language support is provided for writing device drivers for OS/2, FutureWare offers a C language interface to the kernel. Developers can create their own C language interface to assembly language calls using the material in this article. The article assumes that the reader is familiar with how to write device drivers; however, additional background information about device driver programming is available from several sources[1].

Device drivers are conveniently written using the small programming model. The first consideration, proper sequence of the data and code segments, is accomplished by linking a file similar to Figure 1. This file causes the data segment to be linked before the code segment, and creates the special segments LASTDATA and CLOSETEXT.

Segments LASTDATA and CLOSETEXT, defined in Figure 2, contain the definition of the address of function endtext(), and of data variables DevHlp and enddata. The address of endtext() is the length of the driver's main code segment. The address of enddata is the length of the driver's main data segment. These lengths, used at initialization, size the driver's main code and data segments. Variable DevHlp is set at initialization to the address of the device helper routines, so helper services are available after initialization.

Figure 3 defines functions and structures used by the driver, which must appear at the head of the driver file. Strategy entry points



```
;Forces correct sequencing of code and data segments
;
_DATA       SEGMENT     WORD PUBLIC 'DATA'
_DATA       ENDS
_BSS        SEGMENT     WORD PUBLIC 'BSS'
_BSS        ENDS
CONST       SEGMENT     WORD PUBLIC 'CONST'
CONST       ENDS
_LASTDATA   SEGMENT     WORD PUBLIC 'LASTDATA'
_LASTDATA   ENDS
_TEXT       SEGMENT     WORD PUBLIC 'CODE'
_TEXT       ENDS
_CLOSETEXT  SEGMENT     WORD PUBLIC 'ENDCODE'
_CLOSETEXT  ENDS
DGROUP      GROUP       CONST, _BSS, _DATA, _LASTDATA
CGRP        GROUP       _TEXT, _CLOSETEXT
public      __acrtused
__acrtused  equ         0           ;defeat import of "C" routines
end
```

Figure 1. Segments Must Be Correctly Ordered in a Device Driver

[1] Two publications providing information about device driver programming are *Writing OS/2 Device Drivers* by Raymond Westwater (Addison-Wesley, 1989) and *OS/2 Version 1.2 Programming Tools and Information*, available for $650 from the IBM Developer Assistance Program.

must be declared before their first reference in the driver header structure.

The driver header is declared as an array of entries, one per character device. Only a single header is used for a block device driver. The header must be declared before any other data variables, since it must appear at the beginning of the data segment. At link time, if the device header fails to appear at the beginning of the data segment, place the driver header in a separate file, and link it before the driver file.

The function driverentry(), in Figure 4, returns a pointer to the strategy packet. This function must be called at entry to the strategy routine. Otherwise, the address of the strategy packet may be lost because it is contained in registers until the call to driverentry(). A switch statement typically follows the use of function driverentry(). The switch statement is executed on the command code contained in the strategy packet header (field reqcommand defined in structure REQHEADER, as in Figure 3).

Command code 0 represents the initialization function. The strategy routine should set the address of the device driver helper routines into variable DevHlp, and set the length of the code segment, data segment, and the

```
;define address of endtext() to be length of code segment
;define storage for variable DevHlp
;define address of enddata to be length of data segment

        public  _endtext,_enddata,_DevHlp
        assume  cs:_CLOSETEXT
_CLOSETEXT      SEGMENT         WORD PUBLIC 'ENDCODE'
_endtext:
_CLOSETEXT      ends
_LASTDATA       SEGMENT         WORD PUBLIC 'LASTDATA'
_DevHlp         dd              ?
_enddata        label           byte
_LASTDATA       ends
        end
```

Figure 2. Private Variables Used by the C Language Interface

```
void endtext();
extern char near enddata;
extern char far * DevHlp;

struct DRIVERHEADER {                   /* template for driver header */
  struct DRIVERHEADER far * nextheader;
  unsigned daw;                         /* driver attribute word */
  void near * strategy;                 /* offset to strategy entry */
  void near * IDC;                      /* offset to IDC entry */
  char name[8];                         /* eight character name */
  char reserved[8];
  };

struct REQHEADER {                      /* template for request header */
  unsigned char reqlen;                 /* length of request packet */
  unsigned char requnit;                /* unit number for block device*/
  unsigned char reqcommand;             /* command number */
  unsigned reqstatus;                   /* returned status */
  long reqreserved;                     /* reserved for use by OS/2 */
  long reqlinkage;                      /* link word used by queueing routines */
  };

struct INITIN {                         /* format of INIT packet at entry */
  struct REQHEADER initinhdr;           /* allocate strategy packet header */
  unsigned char initinunused;           /* not used at entry */
  char far * initinDevHlp;              /* pointer to DevHlp routines */
  char far * initinInit;                /* pointer to startup command line */
  unsigned char initindrive;            /* first drive number, block device */
  };

struct INITOUT {                        /* format of INIT packet on return */
  struct REQHEADER initouthdr;          /* allocate strategy packet header */
  unsigned char initoutunits;           /* number of units for block device */
  unsigned initoutcodelen;              /* length of code segment */
  unsigned initoutdatalen;              /* length of data segment */
  char far * initoutBPBptr;             /* pointer to BPB list for block device */
  };
```

Figure 3. Function Prototypes and Data Structure Declarations (Continued)

```
/*============================================================*/
/*                  Driver Entry/Exit Routines                */
/*============================================================*/

struct REQHEADER far * driverentry();            /*  Figure 4  */

void driverexit(struct REQHEADER far * reqpkt);  /*  Figure 5  */
        /* set strategy packet into ES:BX for return */
        /* return from driver */

unsigned SetIRQ(void (* near intentry)(), unsigned IRQnum, unsigned
shared);
        /* Figure 6, intentry is name of interrupt function */
        /* IRQnum is number of IRQ to be intercepted */
        /* shared is 0 for private IRQ, 1 for shared IRQ */

void dismiss(unsigned acknowledge);  /* Figure 7  */
       /* perform far return from interrupt service routine */
       /* if acknowledge = 1, interrupt was handled */
       /* if acknowledge = 0, pass interrupt to next handler */

void int3();  /* Figure 8, generate int3 for debugger */
```

Figure 3. Function Prototypes and Data Structure Declarations

status of the initialization operation into the strategy packet.

The pointer to the strategy packet may be cast to type (struct INITIN far *) to copy the address of the device helper routine from the strategy packet into variable DevHlp. Structure INITIN and variable DevHlp are defined in Figure 3.

Cast the pointer to the strategy packet to type (struct INITOUT far *) to set the length of code and data segments, as well as completion status, before exiting the initialization function. Function endtext() and variable enddata are declared in Figure 3. Their addresses, as defined in Figure 2, are the length of the main code and data segments, respectively.

Strategy routines, other than command code 0(initialize), recover the pointer to the strategy packet with routine driverentry(). Before completion, the strategy routine must set completion status in the strategy packet.

Once the strategy routine is completed (all command codes), the driver returns to the kernel with function driverexit(), as in Figure 5, which aligns the stack and performs the far return to the kernel. Any code following driverexit() will not be executed. The address of the strategy packet must be passed as an argument to driverexit().

```
;struct REQHEADER far * driverentry();
;Returns long pointer to strategy packet.  This function
;must be called immediately upon entering the strategy
;routine, or the pointer may be lost.
;
        public  _driverentry

        assume  cs:_TEXT
_TEXT           SEGMENT      WORD PUBLIC 'CODE'
_driverentry:
        mov     dx,es
        mov     ax,bx
        retn
_TEXT           ends
        end
```

Figure 4. Preserve Address of Strategy Packet

```
;void driverexit(struct REQHEADER far * reqpkt);
;use this function to exit from the strategy routine
;
        public  _driverexit
        reqpkt  equ          4
        assume  cs:_TEXT
_TEXT           SEGMENT WORD PUBLIC 'CODE'
_driverexit:
        push    bp
        mov     bp,sp
        mov     es,word ptr reqpkt+2[bp]
        mov     bx,word ptr reqpkt[bp]
        pop     bp
        mov     sp,bp
        pop     bp
        retf
_TEXT           ends
        end
```

Figure 5. Exit Strategy Routine

```
;unsigned SetIRQ(void (* near intentry)(), unsigned IRQnum, unsigned shared);
;allow driver to intercept hardware interrupt
;
        public  _SetIRQ
        extrn   _DevHlp:dword

intentry        equ         4
IRQnum          equ         6
shared          equ         8

        assume  cs:_TEXT
_TEXT   SEGMENT WORD PUBLIC 'CODE'
_SetIRQ:
        push    bp
        mov     bp,sp
        mov     ax,word ptr intentry[bp]
        mov     bx,word ptr IRQnum[bp]
        mov     dh,byte ptr shared[bp]
        mov     dl,1Bh                      ;SetIRQ
        call    [_DevHlp]
        jc      error
        mov     ax,0
        pop     bp
        retn
error:
        mov     ax,1
        pop     bp
        retn

_TEXT   ENDS
        end
```

Figure 6. Establish Interrupt Service Routine

```
;void dismiss(unsigned acknowledge);
;dismiss hardware interrupt by performing far return
;clear carry to indicate service of interrupt, or set
;carry to chain interrupt to next driver sharing
;interrupt level
;
        public  _dismiss
acknowledge     equ     4
        assume  cs:_TEXT
_TEXT           SEGMENT WORD PUBLIC 'CODE'
_dismiss:
        push    bp
        mov     bp,sp
        cmp     word ptr acknowledge[bp],0
        je      setcarry
        clc
        jmp     short donecarry
setcarry:
        stc
donecarry:
        pop     bp
        mov     sp,bp
        pop     bp
        retf
_TEXT           ends
        end
```

Figure 7. Exit Interrupt Service Routine

Interrupts are set using function SetIRQ(), which takes the name of the near function to be called at interrupt time (and other arguments), as in Figure 6. When an interrupt occurs, the interrupt manager sets the CS and DS registers and calls the specified function. Interrupt functions should terminate with a call to function dismiss() (see Figure 7), which aligns the stack, sets completion status to the interrupt manager via the carry bit, and performs a far return. The reason the small model was chosen for all functions is now clear—the strategy and interrupt managers can make calls only into a single code segment. The special routines driverexit(), as shown in Figure 5, and dismiss(), as shown in Figure 7, allow a far return from a near function.

The function int3() in Figure 8 may generate an INT 3 instruction that invokes a kernel debugger, if present.

Compilation of the driver is done with disabling of stack checking and with the "packed structure" directive set:

```
cl /Asnw /G2s /Zp /c sample.c
```

Linkage of the driver requires the file shown in Figure 1 (PROLOG.OBJ) to be linked before all other files. To simplify linking, a library file DEVHLP.LIB is assumed to contain the object modules defining all device helper functions and support functions provided with the interface. Each function is defined in a separate file to minimize the size of code included from the library into the driver.

All device drivers require a .DEF file containing the LIBRARY directive. A sample LINK command is:

```
LINK/MAP/NOD
prolog+sample,sample.sys,,devhlp.lib,
sample.def
```

A sample driver, shown in Figure 9, illustrates the protocols the driver writer must follow to implement the C language interface. Figure 3 shows the definition of functions and structures used by the driver. The driver header is declared as a single instance of structure REQHEADER, as this driver represents the character driver DEMO$. The header must reside before any other data variables in the data segment. As a practical matter, the header is often compiled in a separate file and linked before the rest of the driver file, forcing correct alignment of the data segment.

The strategy routine, strategy(), performs the minimal operations necessary to complete the initialization function. The temporary variable strat is allocated on the stack to hold the far address of the strategy packet. Use of stack space should be minimized. The driver writer can allocate variables on the stack or in the data segment.

```
;void int3();
;execute INT 3 instruction to invoke kernel debugger
;
        public  _int3
        assume  cs:_TEXT
_TEXT           SEGMENT      WORD PUBLIC 'CODE'
_int3:
        int     3
        retn
_TEXT           ends
        end
```

Figure 8. Breakpoint Instruction for Kernel Debugging

```
void strategy();      /* strategy entry points must be declared before /*
                      /* reference in driver header */

#include "devhlp.h"   /* variables DevHlp, endtext and endcode */
                      /* are declared in Figure 3 */

/* driver header declarations must precede all other variable declarations */

static struct DRIVERHEADER dheader = {
   (struct DRIVERHEADER far *)-1,
                      /* pointer to next header */
   0x8080,            /* character driver, OS/2 level 1 */
   strategy,          /* strategy routine address */
   0,                 /* address of IDC routine */
   {'D','E','M','O','$',' ',' ',' ',' '},
                      /* driver name: DEMO$: */
   ""                 /* reserved */
   };

void strategy()       /* DEMO$: strategy routine */
{
   struct REQHEADER far * strat;
                      /* "strat" will be used to hold the FAR */
                      /* address of the strategy packet.  */
                      /* Minimize use of stack in the strategy */
                      /* and interrupt routines */
   int3();            /* Generate INT 3 instruction, to enter */
                      /* kernel debugger */
   strat = driverentry();
                      /* Set "strat" to FAR address of request */
                      /* packet.  Function "driverentry" should */
                      /* be called at entry to routine */
```

Figure 9. Sample Device Driver (Continued)

```
    switch (strat-reqcommand)
       {
       case 0:          /* Command code 0 represents INIT packet */
                        /* struct INITIN represents INIT packet */
                        /* structure at entry to routine.  The */
                        /* INIT packet structure at exit is */
                        /* represented by struct INITOUT */
         DevHlp = ((struct INITIN far *)strat)-initinDevHlp;
                        /* Global variable DevHlp, defined in Figure 3 */
                        /* must be initialized by initialization routine */
         ((struct INITOUT far *)strat)-initoutcodelen =
                 (unsigned)endtext;
                        /* The length of the code segment is the address of
                        /* function "endtext", which is defined in Figure 3 */
         ((struct INITOUT far *)strat)-initoutdatalen =
                 (unsigned)&enddata;
                        /* The length of the data segment is the address of
                        /* variable "enddata", which is defined in Figure 3 */
         strat-reqstatus=0x100;
                        /* Set return status = 100, success */
         break;
       default:
         strat-reqstatus=0x8103;
                       /* Set return status = 8103, error: Unknown Command */
         break;
       }
    driverexit(strat);
                       /* Function "driverexit" is called to return to OS/2 */
                       /* kernel */
}
```

Figure 9. Sample Device Driver

Developers may expand on these routines to produce their own interface.

The function driverentry(), shown in Figure 4, sets the contents of the variable strat. This function must be called at entry to the strategy routine, or the address of the strategy packet may be lost. A switch statement is executed on the command code contained in the strategy packet header (field reqcommand).

Command code 0 represents the initialization function. Variable strat is cast to type (struct INITIN far *) to recover the address of the device helper routine from the strategy packet. The address of the helper service routine is placed into variable DevHlp. Then it can be referenced by subsequent strategy requests or by an interrupt service routine. Structure INITIN and variable DevHlp are defined in Figure 3.

Variable strat is cast to type (struct INITOUT far *) to set the length of code, data segments, and completion status. Function endtext() and variable enddata are defined in Figure 3. Their addresses represent the length of the main code and data segments, respectively.

Once the initialization actions have been completed, the driver returns to the kernel with function driverexit(). Any code following driverexit() will not be executed. The address of the strategy packet must be passed as an argument to driverexit().

Developers may expand on these routines to produce their own interface, or subscribe to the routines provided by FutureWare. These routines are available in a companion disk to *Writing OS/2 Device Drivers*, mentioned in the introduction to this article. A kernel debugger, also available from FutureWare, is installed at boot time as a device driver, providing a debugging environment for all versions of the OS/2 kernel. The debugger includes source code for all the modules and the debugger.

Raymond Westwater, *FutureWare, Inc., 78 Temple Avenue, Suite 15, Hackensack, NJ 07601-6026. Mr. Westwater is President of FutureWare, Inc., a full-service OS/2 consulting firm providing custom device drivers, DOS–OS/2 conversion assistance and training. He has been actively involved in writing OS/2 utilities for the last few years. Mr. Westwater has an MS in Mathematics from C.W. Post University.*

Extended Edition Communications Manager

Optimizing APPC Performance Over LANs

by Pat Scherer

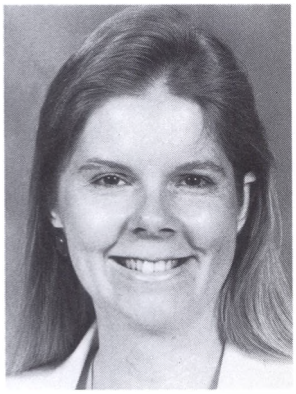
Advanced Program-to-Program Communications (APPC) is an implementation of IBM's Systems Network Architecture (SNA) for Logical Unit (LU) type 6.2. LU 6.2 defines how two or more programs, running on a variety of hardware and using a variety of connectivities, cooperate as peers in performing specified transaction processes. APPC, a major component of IBM's Systems Application Architecture (SAA) strategy, provides a powerful platform for distributed processing across heterogeneous environments. This article provides an overview of APPC features implemented by OS/2 Extended Edition (EE) followed by a description of transaction program design tips for maximizing APPC performance. The last section describes the OS/2 EE Configuration and Attach Manager services with recommendations for optimal performance tuning of the Local Area Network (LAN) and APPC configuration parameters.

APPC

APPC was introduced in the late 1970s as an SNA enhancement that enables cooperative processing between host systems. As personal computers became more powerful in the mid-1980s, peer-to-peer transactions among small systems became feasible. The first APPC offering for the PC environment became available in 1985. This first implementation proved to be functionally rich, but challenging to program and configure. The introduction of an enhanced Application Programming Interface (API), the Attach Manager, and the configuration menus of the OS/2 EE Communications Manager improved usability, which continues to be a major focus for future releases.

A few terms and concepts are central to APPC. A user communicates through APPC by using transaction programs. A transaction program is a sequence of communication commands, called *verbs*, interspersed with routines for handling data that is sent or received. Transaction programs are generally transparent to the end user; they typically are embedded in the user's applications (such as a database program). A transaction program uses APPC's API by providing the name of the APPC verb and other information in a format called a Verb Control Block (VCB) and calling APPC. APPC processes the command on behalf of the application, routing information through the lower communication layers.

(For further information about APPC, see two previous articles in the *Personal Systems Developer*: "Advanced Program-to-Program Communications," Summer 1989 and "The APPC Attach Manager," Winter 1990.)



Pat Scherer

MORE ABOUT TRANSACTION PROGRAMS

To communicate, APPC must first establish a session with the remote system. Many protocols allow sessions to be established only between a local system and a server or host. These are called master/slave protocols. APPC, in contrast, allows direct communication between peer systems. Within APPC, the logical communications port for the transaction program is identified by a Logical Unit Name (LUname). In addition, the LAN adapter has a burned-in or universally administered address. The remote system is identified by a Partner LU Name (PLUname) and an adapter address. When a transaction program starts, it registers the LUname under which it will run. To initiate communication

with a partner, it must provide the PLUname and the transaction program name of the partner. It issues an ALLOCATE verb to APPC. The ALLOCATE verb establishes a conversation on the session, if one is available. If not, ALLOCATE can cause a new session to be activated, depending on the options specified in the ALLOCATE verb. A session may contain more than one conversation, but within OS/2 EE, only one conversation per session can be active at a given time. Multiple sessions within OS/2 EE enable concurrent communication between two or more LUs.

APPC is architected around a scheme known as half-duplex. Every transaction program on the local LU must have a complementary transaction program on the partner LU that can react appropriately to incoming requests. The inquiry transaction in Figure 1 illustrates how these complementary transaction programs interact.

As illustrated in Figure 1, conversations are divided into three sections: conversation startup, data exchange, and conversation take-down. Conversation startup is accomplished by the verbs TP_STARTED, ALLOCATE, and RECEIVE_ALLOCATE. TP_STARTED tells APPC to reserve resources for a new transaction program; APPC then provides a Transaction Program ID. The transaction program ALLOCATEs the conversation, which queues a request for the partner LU. SEND_DATA queues one record to send to the partner LU. The data does not flow over the LAN, however, until a RECEIVE_AND_WAIT is issued. APPC buffers outgoing data until: 1) a receive verb is issued, 2) it is ordered to FLUSH its transmit buffers, or 3) the transmit buffers are full and therefore, must be flushed. These options provide control for the efficient use of buffers and hardware capacity which optimizes performance.

The partner LU's transaction program issues the RECEIVE_ALLOCATE verb. The transaction program may queue a RECEIVE_ALLOCATE before receiving the ALLOCATE request. On the other hand, the APPC Attach Manager may be configured allowing the transaction program to issue the RECEIVE_ALLOCATE after receiving the ALLOCATE request. RECEIVE_AND_WAIT processes the incoming data; SEND_DATA is issued to respond to the inquiry. The DEALLOCATE verb ends the conversation and initiates the transmission of the queued reply. The partner then issues a TP_ENDED, which frees all the resources associated with the transaction.

The local LU completes its first RECEIVE_AND_WAIT when it receives the incoming data (reply). The second RECEIVE_AND_WAIT processes the incoming DEALLOCATE request. The local LU responds to the DEALLOCATE request by deallocating its side of the conversation. It then issues a TP_ENDED, which releases its resources.

TRANSACTION PROGRAM DESIGN TIPS

A complete list of verbs available within the OS/2 EE implementation of APPC is found in the IBM *Operating System/2 Extended Edition Version 1.2 Advanced Program-to-Program Communications Programming Reference*. A new section called "Guidelines for Improving Transaction Program Throughput" has been

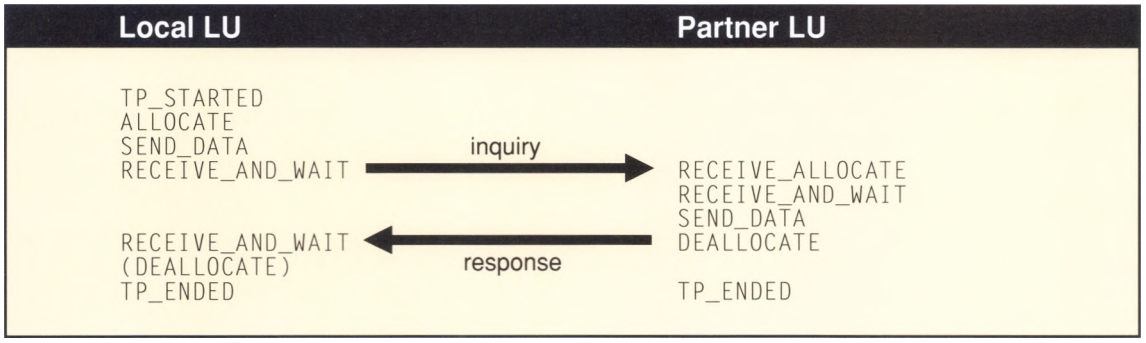


Figure 1. An Inquiry Transaction

added to this publication. The suggestions that follow summarize and explain some of the most beneficial guidelines in this reference. The inquiry example presented in Figure 1 illustrates some of the ideas presented for designing transaction programs for optimal performance.

Use Extended Verbs

OS/2 EE 1.2 provides a set of APPC enhancements called *extended verbs*. A Type parameter that has been added to the SEND_DATA verb allows it to be combined with a FLUSH, CONFIRM, or DEALLOCATE verb, thus eliminating the need for a second command. Likewise, RECEIVE_AND_WAIT has been modified to process the enhanced SEND_DATA with a single verb. Using the extended verb set results in a protocol that is identical to the previous verb set. The extended verb set can be used for transaction programs communicating with other APPC transaction programs that do not support the verb extensions. The extended verb set does not reduce the amount of work APPC has to do, but it does improve the performance by reducing the overhead associated with the additional API crossings between APPC and the transaction program. Figure 2 shows the inquiry transaction using extended verbs.

Note the use of SEND_DATA (FLUSH) in Figure 2. FLUSH causes an immediate transmission of data present in the buffers. Using the FLUSH parameter on the SEND_DATA results in a slight performance improvement. That is because the local LU issues the RECEIVE_AND_WAIT after the inquiry is sent on the LAN. FLUSH should be used judiciously. In many cases, it can slow performance by causing extra data transmissions and suboptimal use of the buffers. FLUSH would not be beneficial, for example, when multiple SEND_DATAs are issued. APPC normally would fill the buffers before sending. If the maximum framesize was 2 KB and each SEND_DATA was 1 KB long, flushing the buffers would double the number of transmissions, and leave the buffers only 50% utilized.

Minimize the Use of CONFIRM

CONFIRM is a request for acknowledgement from the partner upon receiving data or a DEALLOCATE request. CONFIRM should be issued only when a confirmation is necessary: it forces the local transaction program to wait for receipt of the acknowledgement and causes extra processing time for both LUs. In cases similar to the inquiry example, receipt of data from the partner LU implies receipt of the inquiry from the local LU; therefore, an explicit confirmation is unnecessary.

Minimize the Use of SEND_ERROR

SEND_ERROR notifies the transaction program at the partner LU that an error has been detected. It should be used only when an actual error is detected, not within the normal Send/Receive path, because it causes two additional message exchanges with the partner and writes an entry to the Communications Manager error log.

Minimize the Use of REQUEST_TO_SEND and PREPARE_TO_RECEIVE

Within APPC, only one LU has permission to send at a given time. This status is called the *conversation state*. REQUEST_TO_SEND is sent to the partner LU to get permission to

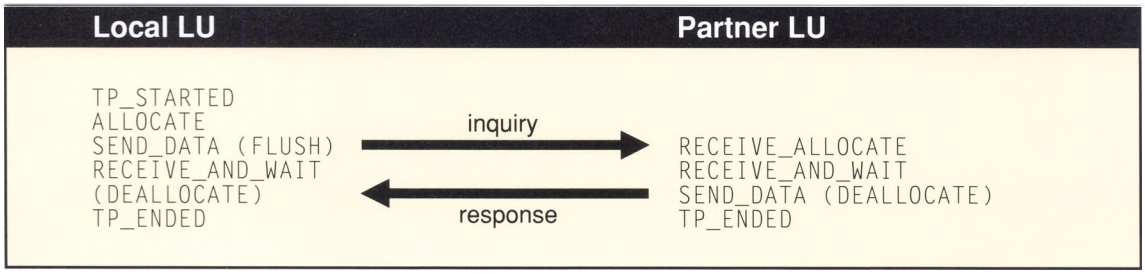


Figure 2. An Inquiry Transaction (OS/2 EE 1.2 Extended Verbs)

send. PREPARE_TO_RECEIVE, used by the sender, reverses the conversation state so the sender becomes the receiver and the receiver becomes the sender. Ideally, transaction programs should keep track of their conversation state and exchange their Send and Receive states when appropriate. If a sender issues a RECEIVE_AND_WAIT when it has finished sending data, its partner is not forced to get permission to send.

Consider Using Basic Conversation

The OS/2 EE APPC API provides verb sets supporting two types of conversation: *basic* and *mapped*. The verbs shown in the inquiry flow are from the basic conversation set. "MC_" preceding the verb indicates that the conversation is mapped, making it easier to use since it was designed for use by applications. Basic conversations are more flexible, but are used by system services. In a mapped conversation, MC_SEND_DATA sends a single record each time it is issued. Basic conversation allows a Logical Length (LL) field of up to 64 KB to specify the total length of multiple records to be sent with a single SEND_DATA command. This is used primarily for transferring blocks of data or files. Performance can be further improved if the logical length is fixed and does not require recalculation by the transaction program.

Minimize the Number of RECEIVE_DATAs

When receiving large blocks of data or files, as much data as possible should be received on each Receive verb. If adequate memory is available, allocate a large memory block for incoming frames. If using Basic Conversation verbs, specify "fill buffer" on APPC Receive verbs rather than "fill ll"; this allows a single Receive verb to handle multiple records.

Use DEALLOCATE (FLUSH)

The DEALLOCATE with type (FLUSH) supplied parameter is faster than the other types on the DEALLOCATE verb. Using DEALLOCATE with one of the ABEND types causes two internal message exchanges with the partner. A third message is written to the Communications Manager error log. Depending on the resources available on the partner system, if APPC is used frequently, and there is no contention for a session, a conversation may be kept active and then DEALLOCATED once the transaction program is finished. DEALLOCATE should always be issued before ending the transaction program. If TP_ENDED is issued before deallocating the conversation, a DEALLOCATE (ABEND) will be issued, with the same performance ramifications described above.

Reuse Sessions Using TP_ENDED (SOFT)

TP_ENDED (HARD) causes all sessions being used by the transaction program to be taken down. If these sessions are to be used again, they will need to be brought back up. It is much more efficient to reuse sessions. TP_ENDED (SOFT) does not take down the sessions and, therefore, allows them to be reused.

Convert Only Once

The transaction program uses a utility verb called CONVERT to convert certain fields (such as TPNAME and MODENAME) from ASCII to EBCDIC. These fields should be converted at the beginning of the transaction program and the results saved for later use.

Recycle Buffers and Verb Control Blocks

Each different segment used for a data buffer incurs extra APPC overhead; therefore, reuse the same segments whenever possible. Several fields in the Verb Control Blocks, such as TP_ID, CONV_ID and conversation type, remain constant throughout a conversation and should not be set to zero or reentered. In addition, fields used exclusively as "returned parameters" (such as the primary and secondary return code fields) do not need to be set to zero.

Use OS/2 Concurrent Processing Capabilities Wisely

OS/2's concurrent processing capabilities can be very effective. Note, however, that each thread created incurs overhead (to create the thread and for each task switch). Semaphores used to synchronize the activities on threads also incur overhead. Unless there is a high potential for process overlap, such as when pipelining disk I/O with transaction processing, the simple approach is usually the best approach. Thread priorities must consider the priorities of all other activities running. In general, the time-critical priority class should be avoided since it may interfere with other system and communication services.

OS/2 EXTENDED EDITION CONFIGURATION MENUS

OS/2 EE provides configuration menus and base configuration files for use by its Communications Manager. The configuration files provide information to the Communications Subsystem, including what communication adapters are installed, what local transaction programs can be called by what partner LUs, and how much data can be sent or received. The configuration menus provide a structure for changing the configuration files, as well as online helps and error detection.

In tuning OS/2 EE for optimum performance, the objective is to maximize the amount of data to be processed at each layer without wasting memory. To accomplish this, the default values in the OS/2 EE base configurations were carefully selected. This objective is challenging, however, because different environments and uses of APPC and LAN imply different performance-tuning considerations. The defaults used by OS/2 EE, for example, are designed to be compatible with older IBM LAN adapters (adapters that do not take advantage of the added capacity of a Token-Ring 16/4 Adapter), They are tuned for "mixed traffic" of 96- to 1048-byte frames. Customizing the APPC/LAN configuration can produce substantial performance gains if any of the following conditions apply:

- The minimum frame size exceeds 96 bytes
- Optimal performance is needed either for interactive communications (such as database queries) or for large file transfers
- The adapter contains more than 16 KB of shared RAM

Recommendations for reconfiguring APPC and LAN are presented later.

OS/2 EE 1.2 provides APPC over Synchronous Data Link Control (SDLC) and the following LAN connectivities: Token Ring, PC Network, and Etherand. When OS/2 EE is installed with the APPC/LAN options, a device driver statement like the one below is placed in the CONFIG.SYS file:

```
device driver=acstrnet.sys
config=c:\cmlib\acscfgus.cfg
```

The ACSCFGUS.CFG file is the default configuration file provided by OS/2 EE for use by APPC and LAN Manager for APPC over LAN communications. This file should be copied to a new file, which can then be modified. Changing the device driver CONFIG= statement in CONFIG.SYS to the name of the new configuration file enables the system to use the customized parameters of the new file when it is booted. Figure 3 shows the structure of the configuration menus used for LAN and APPC. The rest of this article covers only those parameters that are useful for performance tuning. For more comprehensive description of all the configuration parameters, see the online Help menus and the IBM *OS/2 Extended Edition Version 1.2 System Administrator's Guide*.

CONFIGURING APPC AND LAN FOR OPTIMUM PERFORMANCE

The example of a 100 KB file transfer over a Token-Ring LAN (shown in Figure 4) illustrates how the configuration parameters affect data flows, and eventually, performance. The diagram indicates the param-

eters affecting the number and size of packets of data flowing through each layer of the local and partner LU. It also illustrates the terminology used to describe data at various points in the process. The Data Link Control (DLC) within OS/2 EE has a mapping layer between APPC and the chosen connectivity, not a true DLC in the architectural sense.

The following summarizes the process illustrated in Figure 4: Before sending data, the configured Request/Reply Unit (RU) size is negotiated with that of the partner to determine the maximum RU size to be exchanged. SEND_DATA fields are broken into RUs that flow to the LAN where a link header is added, converting the RU to a frame. The frame is placed into a transmit buffer and sent over the LAN. When the data reaches the partner, the frame is stripped of its header. The resulting RU is passed to APPC.

Transaction programs usually fall into one of two categories: interactive transactions (such as the inquiry example) and file transfers. While some tuning recommendations improve the performance of both types of transactions, many favor one type over another. Before making changes to the APPC/LAN configurations, users should decide which type of transaction to prioritize and identify any resource constraints (such as limited RAM on the processor or card). The following sections describe the primary performance considerations within the APPC/LAN configuration and alterations for parameters to improve performance.

DLC Profile

The maximum RU size should be matched to the maximum RU size configured in the mode. The Communications Manager configuration verification routine does not allow the DLC's maximum RU size to be smaller than that of the mode. Although the verification routine allows the DLC maximum RU size to be larger than the value configured in the mode, this wastes buffer memory that may be needed to handle high levels of traffic.

The Free Unused Links parameter conserves LAN and OS/2 resources when a link that has been activated is no longer being used. The overhead associated with maintaining a single link is small, but maintaining many unused links may degrade performance. If, on the other hand, a small number of links

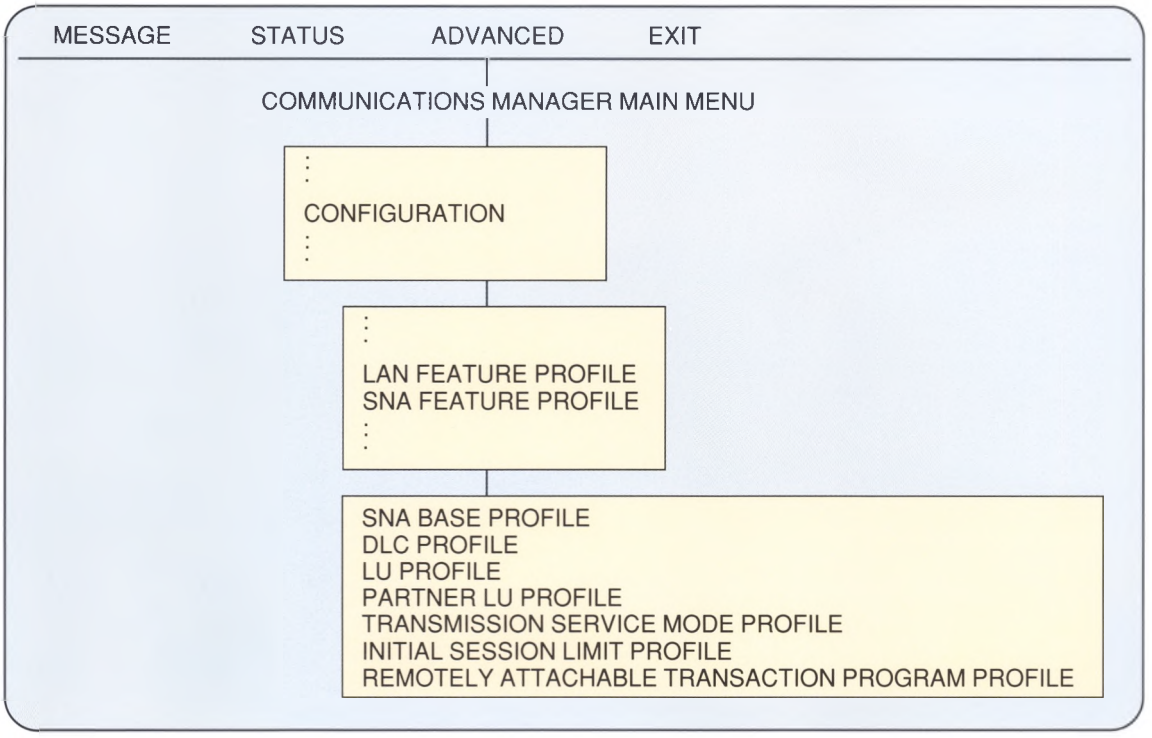


Figure 3. APPC/LAN Configuration Menu Tree

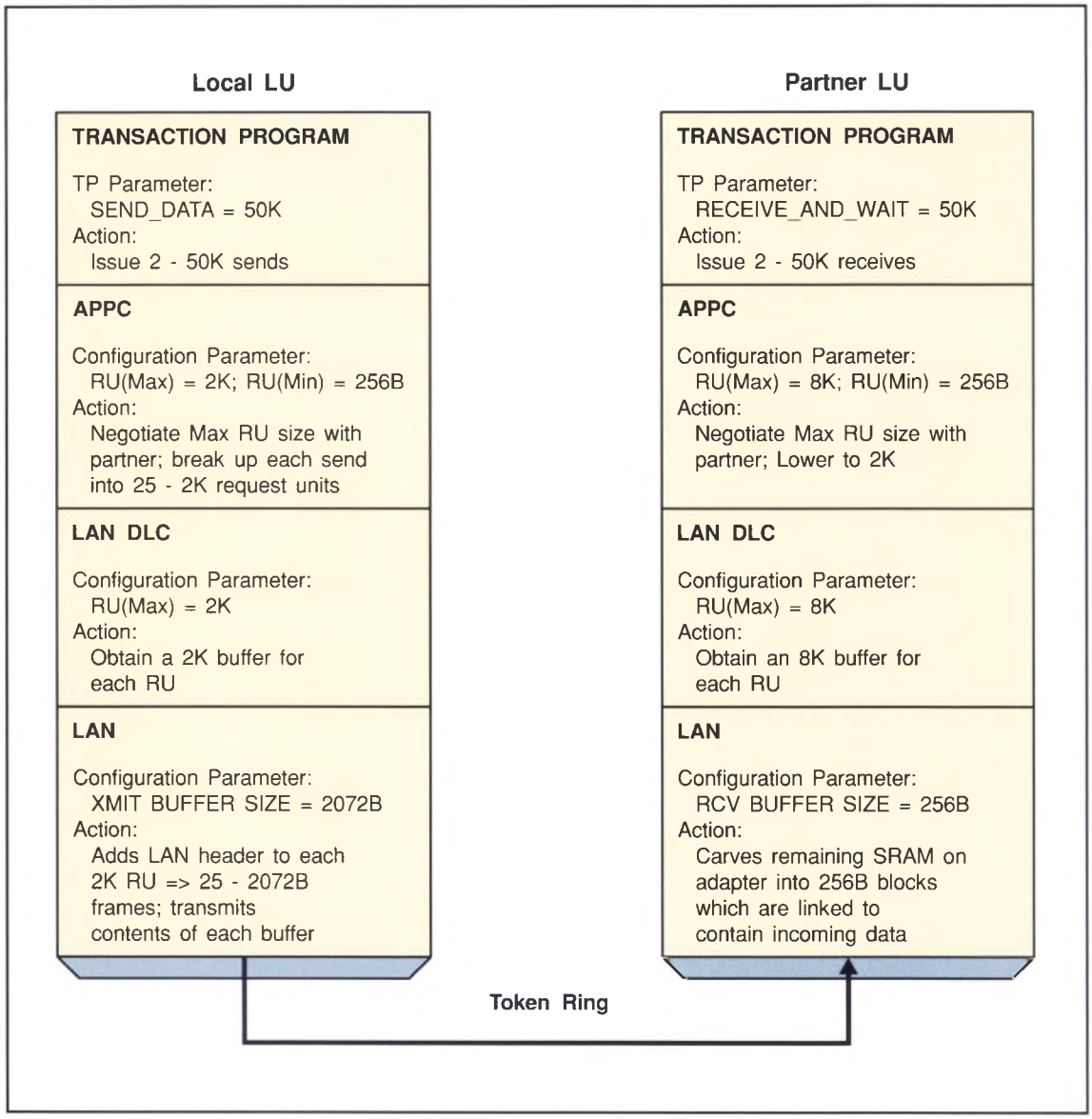


Figure 4. Data Flow Example for a 100 KB File Transfer

are used frequently, the time can be saved by not freeing unused links.

The Send Window Count controls how many frames are sent before pausing to wait for a hardware acknowledgement from the partner. The Send Window Count should always be larger than the partner's Receive Window Count. If the Send Window Count is less than the partner's Receive Window Count, the LU waits for acknowledgment from the partner after sending its send count of frames. Since the partner is expecting more frames than were sent, the transaction hangs until a timer prompts the partner to acknowledge receiving the data. To prevent this problem, the local LU negotiates the Send and Receive Window Counts with the partner before setting the working values. For interactive transactions between PCs and PS/2s, the defaults of Send Window Count=2 and Receive Window Count=1 are optimum. File transfers benefit from larger settings of Send Window Count=5 and Receive Window Count=4. Most mainframes use a set Send and Receive Window Size; the DLC resets the parameters on the PC or PS/2 to match those of the mainframe during negotiation.

Transmission Service Mode Profile

If file transfer throughput is a priority, it is best to use the largest maximum RU size supported by the local and frequently used remote adapters. That means setting the maximum RU to 1500 bytes if using an Ethernet Adapter, 1920 bytes if using an IBM Token-Ring Adapter or Token-Ring Adapter/A, and as high as 15128 bytes if using Token-Ring 16/4 Adapters. APPC negotiates with the partner LU to establish an RU size that both LUs can support. Setting a very large RU size on a server, for example, is not beneficial unless other systems can support the same RU size.

The Mode Pacing parameter specifies the number of RUs that may be sent before a session-level pacing response is required. The default setting of Pacing=8 is optimized for moderate to heavy traffic using multiple links. If transactions are largely interactive or if the LAN is lightly loaded, better response results from raising the pacing window to 16.

LAN Feature Profile

The Transmit Buffer Size should be set to the maximum RU size + 24. The optimum Receive Buffer Size is the largest configurable value equal to or less than the transmit buffer size which provides a minimum number of receive buffers adequate to handle incoming data. Configurable values for Receive Buffer Size are between 96 and 2048 bytes in 8-byte increments. The minimum number of receive buffers required to handle incoming data is ascertained by the following formula:

```
# Receive Buffers = (Transmit Buffer
Size * 2) / Receive Buffer Size
```

If this value overcommits the available RAM, the error is reported when verifying the configuration, and the buffer sizes will need to be reduced. Otherwise, all available RAM is divided into receive buffers of the configured size.

CONCLUSION

APPC provides the tools necessary for reliable, efficient communication across a wide variety of environments. OS/2 EE has made great progress toward simplifying its APIs through its Attach Manager Services and menu-driven configuration program. We can expect that both performance and usability will continue to improve as APPC matures.

ACKNOWLEDGEMENTS

To John Q. Walker and Greg Callis, whose work formed the basis for the Performance Section of the *OS/2 Extended Edition 1.2 APPC Programming Reference*.

REFERENCES

IBM. *OS/2 Extended Edition Version 1.2 Advanced Program-to-Program Communications Programming Reference*. (SC23-0959).

IBM. *OS/2 Extended Edition Version 1.2 System Administrator's Guide*. (S01F-0261).

Pat Scherer, *Austin Programming Center, IBM Entry Systems Division, 11400 Burnet Road, Austin, TX 78750. Ms. Scherer, currently a performance lead analyst for OS/2 EE Communications Manager, joined IBM in 1980 as an industrial engineer. She holds an MBA in Industrial Management from North Texas State University.*

Software Tools

Tool Selection Techniques

by Brian Proffit

This article summarizes an IBM Field Television Network broadcast devoted to the selection of Presentation Manager (PM) development tools. Product selection considerations are discussed, as is a technique for evaluating these considerations and arriving at a decision.

Application development tools have significant impacts on development organizations. The tools selected affect programmer productivity, as well as the amount of time (and expense) required to maintain applications. Tools may also determine what additional runtime routines are required by an application, and potentially what royalties are paid to the tool's owner. All of these affect an application's profitability. The tool may also affect the program's performance and usability, which in turn affects the product's acceptance in the market.

Brian Proffit

TOOL SELECTION CRITERIA

There are many considerations when making such an important decision. Figure 1 shows some of the criteria that should be considered in selecting tools.

- ☐ **Documentation**: Is it complete, well organized, well indexed, and easy to read? Are reference cards or keyboard templates included?
- ☐ **Education**: Are classes available? Online tutorials? If the tool's own user interface is CUA-conforming, it should be easier to learn.
- ☐ **Evaluation Copy**: Can you get a copy to review before purchasing the product? Is there a demonstration version on IBM's ESDTOOLS disk that can be downloaded and reviewed?
- ☐ **Target User**: Does the tool suit the target user? The requirements for an accounting department's report-generating tool, for example, will differ considerably from those of experienced C programmers, just as a C programmer's needs will differ from those of a COBOL programmer.
- ☐ **Ease-of-Use**: In addition to the tool's own usability, will it make the final application easier for your customers to use? Does it support features such as direct manipulation and the PM clipboard?
- ☐ **SAA Conformance**: Does the tool produce applications that conform to Systems Application Architecture's Common User Access (CUA) and Common Programming Interface (CPI) guidelines? These features can make the application easier to port to other platforms, more consistent with other applications, and can reduce the learning curve for the new user.
- ☐ **Scope**: Is the tool useful for only one phase of the development process, or does it cover the entire cycle from design and prototyping through development (including documentation), and into maintenance? Will it allow the developer to access other familiar tools across phases of the cycle?

Figure 1. Tools Evaluation Categories (Continued)

- ☐ **Methodology**: Is the tool based on procedural or object-oriented programming techniques? If it is different from what has been used in the past, what education is required for the new methodology? If object-oriented, can the developer import class libraries and objects that already exist? Does the tool use a proprietary language or a standard one? How much education does the new language require?
- ☐ **Prototyping**: Are there easy, interactive prototyping facilities? How much of the prototyped work can be used in the production product? Must the prototyper be shipped with the production product?
- ☐ **Interpretive vs. Compiled**: Is a runtime module required? Is a royalty required for the distribution of the module? How much of the interpreter is required at runtime? Is it implemented in Dynamic Link Libraries (DLLs) to minimize memory requirements? Does it include its own compiler, or require the purchase of a separate one? If a compiler is included, is it an optimizing compiler?
- ☐ **Debugging Abilities**: Does the product include a debugger, or does it require a separate package such as CodeView® or MultiScope?
- ☐ **Look-and-Feel**: Can the generated program exploit new features of OS/2 1.2 and the *CUA Advanced Interface Design Guide,* such as spin buttons, combo boxes, multiple-line entry, multiple document interface, horizontal scrolling of list boxes, proportional sliders in scroll bars, and the Information Presentation Facility (IPF)?
- ☐ **Functionality**: Does it support a wide range of devices (for example, the 1024 x 768 mode of the IBM 8514 display)? Does it exploit OS/2's Dynamic Data Exchange (DDE), the PM clipboard, Structured Query Language (SQL), non-standard PM controls, double-clicking, secondary windows, input data validation, multitasking, Inter-Process Communications (IPC), progress indicators, connectivity to other DLLs or EXEs, Local Area Network (LAN), and metafiles for data exchange? Is support provided for all resource types (icons, dialog boxes, fonts, menus)? Is the function provided directly in the tool, or must the developer write extensions? Can the developer extend the tool, if necessary?
- ☐ **Vendor**: Do vendor and tool participate in IBM's AD/CYCLE family of application development products? How is the product supported? How are fixes distributed? Is there a toll-free number or online Bulletin Board System (BBS) for support, exchange of technical information, and distribution of fixes? Does the vendor participate in existing online services such as CompuServe®? Does the product have a user group?
- ☐ **Migration/Compatibility**: Are there techniques for importing existing programs? Is an upgrade required for new OS/2 releases? Would that affect any versions of the runtime that has been distributed?
- ☐ **Multi-Platform Capability**: Are compatible versions of the tool available for Windows, Macintosh, or X-Windows? If so, are programs restricted to a "least common denominator" of function across those platforms? Can the developer easily customize versions for other platforms to exploit any enhanced function?
- ☐ **National Language Support**: Does the product support languages other than U.S. English? Is the OS/2 codepage support used? Is there support for double-byte languages, such as Japanese? For more information, see the "OS/2 National Language Support" and "National Language Support for the Dialog Manager" articles in this issue.
- ☐ **Prerequisite Products**: Are other products required, such as the OS/2 Toolkit, for resource editors, or an IBM or Microsoft compiler containing CodeView?
- ☐ **Financial**: Is the tool reasonably priced? Are runtime royalties involved? If so, are quantity discounts available? Is site-license pricing offered?

Figure 1. Tools Evaluation Categories

AN EVALUATION TECHNIQUE

Obviously, that is a lot of things to consider! Multiply that by the number of products now available, and it can appear to be an overwhelming task to select one. There is a large amount of information that has to be organized and evaluated into a rational form. Fortunately, you already have a tool which is excellent for that purpose: that is precisely the sort of thing your computer does very well. There are "decision support" packages available, but I tend to apply Occam's Razor (commonly known as the Keep-It-Simple rule) and use a more basic technique. A spreadsheet can be used to provide a clear and concise summary of the various product features and their relative importance (to you). It can help to identify the best product, and to document the reasoning behind the decision.

Begin by developing the set of criteria for making the decision. I recommend dividing the list into two parts. The first can be general points that apply regardless of the type of tool being selected; for example, usability and CUA compliance. The second part consists of items that are specific to the functionality of a particular type of tool. For editors, these might include multiple files in multiple windows, multiple views of the same file, nesting assistance, and language syntax support.

Describe these criteria in the first column of the spreadsheet (column A in Figure 2). Next, think through each criterion's relative

Begin by developing the set of criteria for making the decision.

	A	B	C D	E F	G H
01	Criteria	Importance	Editor_1	Editor_2	Editor_3
02					
03	GENERAL REQUIREMENTS				
04					
05	CUA Compliance	7			
06					
07	Education				
08	– Online Tutorial	4	4	9	3
09	– Print Tutorial	7	9	7	9
10					
11	Usability				
12	– For Novices	2	2	2	3
13	– DP Professional	9	8	7	8
14					
15	Installation				
16	– Simplicity	3	4	6	7
17	– Configuration Changes	7	9	7	8
	⋮				
38	TOOL–SPECIFIC NEEDS				
39					
40	Language Syntax Assist	8	4	7	9
41	– Multiple Languages	4	4	4	10
42					
43	Command Macros	7	9	9	9
44	PM Clipboard				
45	Multiple Windows	9	7	5	8
46	Delimiter Matching	8	7	7	4
	⋮				
69	SCORE		38.6	41.4	47.9

Figure 2. A Sample Spreadsheet for Evaluating Tools

OS/2 tools are listed in the OS/2 Application Guide.

importance to your specific needs. Assign each one a weight from 1 to 10, and put that in the second column (B). These values are based on the type of selection being made. For example, if a program generator is to be used by the accounting department to create reports, usability for novices would be rated as more important than usability by a DP professional. For a programmer's editor, however, those would be reversed. It is unlikely that non-technical people will be developing source code, but usability by a professional programmer is extremely important. It is critical that sufficient time be spent on these weight prioritizations—they form the basis of the evaluation.

Place a heading in row 1 of the spreadsheet for each of the packages you are evaluating, from column C to the right. To simplify the process, leave a blank column to the right of each product. For example, if you were looking at editors, you could put Editor_1 in column C, Editor_2 in column E, Editor_3 in column G, and so on. Under each product's name assign each selection criterion a score from 1 to 10, based on how that product compares with others. If you believe the multiwindow flexibility in Editor_3 is better than that in Editor_1, you might assign an 8 to Editor_3 and a 7 to Editor_1. However, if you do much mainframe work, the XEDIT compatibility seen in the Editor_1 package may cause you to rate its usability higher than that of Editor_3.

Next, in the alternating blank columns (D, F, and so on), calculate the weighted rank of each criteria for each package, using a simple formula that multiplies the criteria's weight times the product's score. Finally, use the last row for the final score for each package, calculated by adding all of the weighted values (that is, columns D, F, H) and dividing by the number of criteria listed.

The result is a score that shows how each package relates to its competitors based on your criteria. If the final scores differ by ten percent or more, your decision is clear. Differences in the five percent range are less critical, but still indicate strong preferences. Differences under five percent are inconclusive, and you should probably select the package with which you "feel" most comfortable.

RESOURCES

In order to evaluate these tools, of course, you have to know about them. A good first step toward finding OS/2 tools is the *OS/2 Application Guide* (IBM publication number G362-0002). It lists dozens of tools that are currently available as well as software that has been announced, but is not yet available. The information in this guide can also be accessed through the IBM Bulletin Board System (BBS) in Atlanta at (404) 835-6600. This BBS supports standard dial-up speeds from 1200 to 9600 bps, and the XMODEM protocol for downloads.

Detailed information about packages you have identified as potentially appropriate can be obtained by calling the company responsible for the software, which is listed in the *Application Guide*. Additionally, members of our Developer Assistance Program recently received a letter describing the ESDTOOLS system, which has demonstration versions of several OS/2 development tools. Many OS/2 tools and utilities developed by IBM programmers for internal company use are now being made available to Developer Assistance Program members. These are yours to download at no charge. OS/2 vendors are being encouraged to make demonstration versions of their products available on this system, and IBM will continue to add more packages as they become available.

SUMMARY

Selection of a tool is a very personalized decision. The next time you read a review of a product, keep in mind the list of criteria *you* would use to select that type of product—it may well be different from the reviewer's list.

Brian Proffit, *IBM Entry Systems Division, 1000 NW 51st Street, Boca Raton, FL 33429. Mr. Proffit is an advisory programmer in OS/2 Software Developer Strategy. He was the lead planner for OS/2 SE 1.1 and is currently responsible for OS/2 developer tools. He joined the development laboratory in Boca Raton in 1983 after working as a systems engineer in Dallas.*

Software Tools

MKS Toolkit for OS/2

by Randall Howard

MKS Toolkit, a collection of over 150 commands and utilities, can increase the productivity of all users, particularly those who are developing software for OS/2. By reproducing the most popular UNIX commands, MKS Toolkit brings all the power of UNIX to the OS/2 developer. Based on the POSIX.2 standard, MKS Toolkit offers the VI full-screen text editor, the AWK Programming Language, the MKS KornShell command interpreter, and other software; it is the perfect bridge between OS/2 and UNIX/POSIX systems. Since a fully compatible version is available for DOS as well, MKS Toolkit is ideal for users who want the same working environment on all their systems.

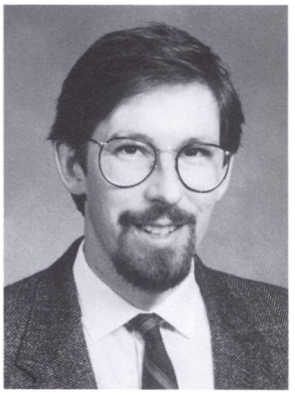
The better the tools, the more productive a programmer can be. To MKS, "better" does not mean just high quality software; it means familiar, mature, standard tools available in a completely integrated software package. Because MKS Toolkit reproduces the best of POSIX (UNIX) software tools, it makes the developer's job easier in several ways:

- Developers can use the same tools on OS/2 that they use on UNIX systems. Since MKS Toolkit is also available for DOS, it offers the familiarity and convenience of the same commands, the same data files, and the same shell scripts on all the development systems.

- The UNIX tools have been developed and refined for two decades. They now constitute a package of commands that do what programmers want. UNIX has always been a programmer's system—designed to provide big power from simple tools.

- UNIX systems are being standardized under the POSIX family of standards, and MKS Toolkit is compatible with POSIX. This guarantees that MKS Toolkit tools will be found on a wide variety of systems; no other set of tools can match POSIX's portability.

The U.S. Government recently announced that POSIX.1 is now a Federal Information Processing Systems Standard (FIPS 151). POSIX.2, which has already been made a draft FIPS standard, is expected to be made a full standard soon.



Randall Howard

BACKGROUND

MKS Toolkit was originally developed for DOS. We did not have the tools we needed to develop DOS software—tools such as profilers to find bottlenecks in programs, programmer-oriented text editors, and a command interpreter with programming capabilities—so we wrote them ourselves. It seemed obvious to follow the lead of UNIX, a well-established system that has been recognized for twenty years as the state of the art for software development.

The experiment was successful: we found that UNIX tools bring the same productivity to non-UNIX systems as they do to UNIX itself. Therefore, it was natural to follow this pattern for OS/2.

The interface for MKS Toolkit was completely re-architected for OS/2, to take advantage of OS/2's powerful facilities. We were able to implement various alarms (such as time-outs) using threads instead of less-sophisticated DOS mechanisms.

MKS Toolkit reproduces the best of POSIX (UNIX) software tools.

CHALLENGES

Bringing MKS Toolkit to OS/2 also provided several challenges. While many OS/2 services are comparable to those of UNIX (like the designers of most modern systems, OS/2 designers learned much from studying UNIX), OS/2 implements some of those services in different ways. Reconciling the different approaches took time and ingenuity.

For example, UNIX begins processes with a "fork" operation. In effect, a process duplicates itself completely; then one copy of the process continues with normal operation while the other goes in a different direction, often invoking other programs and utilities. OS/2 process creation is designed differently. A call that mimicked fork was necessary, however, for the subtleties of implementing a UNIX command interpreter (usually called the shell).

Implementing a true fork on OS/2 proved difficult. While many OS/2 facilities almost worked the way we wanted, none actually did the job. In the end, we settled on the following approach. The original process creates a new segment and copies its data segment into the new one. The process then invokes itself again, creating a new process from the original executable file and passing a special option flag to indicate that a fork operation is in progress. The original process pipes information to the new process, telling it the location of the duplicate data segment. Finally, the new process copies the contents of the duplicate data segment into its own data segment, finishing the cloning operation. (The duplicate data segment cannot be used directly because certain system Dynamic Link Libraries (DLLs) expect to use the original data segment for the new process rather than the one received from the parent process.)

COMPATIBILITY

Why did we implement the fork operation when OS/2 already provided a method for process initiation? Because the fundamental principle of POSIX compatibility underlies all MKS software. We firmly believe that POSIX will become a dominating influence in the computer industry of the 1990s. Indeed, this is already taking place; vendors are scrambling to offer POSIX on almost every proprietary system. MKS already has a proven track record in offering POSIX-compatible software.

MKS Toolkit's compatibility with POSIX is already being put to the test. Many companies need to run the same shell scripts (batch programs) on a variety of systems: their POSIX systems, DOS, and OS/2. Without MKS Toolkit, such companies simply could not use OS/2.

MKS KORNSHELL

The MKS KornShell command interpreter is an important MKS Toolkit command that provides the "look and feel" of POSIX on OS/2. The MKS KornShell works under the Presentation Manager (PM) or MINSHELL. Like the standard CMD.EXE command interpreter, it allows wild card characters in file names, but on a much wider scale; they can be used in directory names as well as file names, for example. New constructs provide more selectivity. For example, "cp [a-m]* a:" copies all files whose names begin with the letters 'a' through 'm' to the diskette in drive A.

Of course, MKS KornShell lets users combine commands using redirection of output and pipes. Since MKS Toolkit commands are designed for productive use in pipelines, piping provides significant power. For example, a single command line such as "find . -type f | cpio -oc | dd of=a:" backs up all the files in the current directory and its subdirectories. The CPIO command squeezes these files into a single file to save space, and the DD command writes directly to the disk in drive A.

Command substitution offers another way to combine commands for power. Any command line can contain the construct, $(command). The command is executed, and the output from the command is then inserted at that point in the command line. As an example, the FIND command displays the names of all the files under a directory that have a certain property; the RM command removes (deletes) files. Thus the command "rm $(find dir -name "*.bak")" removes all files under dir that have the '.bak'

suffix. This includes all files in all subdirectories, sub-subdirectories, and so on; so the command is much more powerful than the simple "rm *.bak."

MKS KornShell also lets users define aliases. Using aliases, short forms can be created for frequently executed longer commands or command sequences. Not only does this save time and trouble when entering a command, but it reduces the number of things to remember. For example, an alias such as "compile" can be created for a standard compiler command line, including all the compiler options normally used. Once the alias is created, type COMPILE FILE to compile a particular file with those options. (Of course, this depends on the command line syntax expected by the compiler.)

Figure 1 shows a sample MKS KornShell function definition. This function works with other functions (not shown) to keep a history of the current directories used in an OS/2 session and to simplify the job of returning to one of those directories at a later point in the session.

THE AWK PROGRAMMING LANGUAGE

MKS AWK is another important part of MKS Toolkit. AWK is a fourth generation language ideally suited for report generation and rapid prototyping. Those who know the C programming language will find AWK very familiar. An AWK program reads through a given data file and performs operations on the records and fields in that file. Figure 2 shows five short (but complete) AWK programs that perform useful data manipulation. Although AWK can do much more, this example illustrates how easy it is to get significant results with a little code.

OTHER IMPORTANT SOFTWARE

Other utilities of special interest in the MKS Toolkit are the VI full-screen text editor, with full macro capabilities, support for National Character sets, and easy global changes; the diff program, which compares files and prints their differences; a spell-checker with a

AWK is ideally suited for report generation and rapid prototyping.

```
function _pwd {
        typeset -i fr to n
        case $# in
        0)   fr=0; to=0;;
        1)   fr=$1; to=0;;
        2)   fr=$1; to=$2;;
        *)   print -u2 "Usage: $0 [-# [-#]]"
             return 2
        esac
        if ((fr<0)); then ((fr=-fr)); fi
        if ((to<0)); then ((to=-to)); fi
        if ((fr<to)); then let n=fr fr=to to=n; fi
        set $PWDS
        n=$#
        if   ((n<=to))
        then return 1
        elif ((fr==to))
        then let n=n-fr; eval print -R '"${'$n'}"'
        else typeset wd
             for wd
             do ((n=n-1))
                if ((fr<n)); then continue; fi
                if ((n<to)); then return; fi
                print "$n $wd"
             done
        fi
}
```

Figure 1. A Sample MKS KornShell Function Definition

```
#Print total number of input lines
END { print NR }

#Print number of lines that contain "string"
/string/ { ++nlines }
END      { print nlines }

#Print line with largest number at beginning
$1 > max  { max = $1; maxline = $0 }
END       { print max, maxline }

#Print sum of all numbers in a file
     { for (i = 1; i <= NF; i++) sum = sum + $i }
END  { print sum }

#Print number of times each word appears in a file
     { for (i = 1; i <= NF; i++)
          wordlist[$i]++; }
END  { for (x in wordlist)
          printf "%s appears %d times\n",x,wordlist[x] }
```

Figure 2. Several Short AWK Programs

dictionary of nearly 100,000 words; various data compression utilities; and the GREP commands, which provide quick searches for character strings or regular expression patterns.

THE MKS PROGRAMMING PLATFORM

MKS offers several other software packages designed to work productively with MKS Toolkit. These include the MKS Revision Control System (RCS), MKS MAKE (for automatically updating programs and keeping files synchronized), and MKS LEX & YACC (compiler construction tools).

MKS software is available from most major software resellers or by contacting MKS at Mortice Kern Systems, 35 King Street North, Waterloo, Ontario, Canada N2J 2W9. The order line is (800) 265-2797 within the continental U.S., and (519) 884-2251 elsewhere. MKS Toolkit is $349.00 (U.S.) for an OS/2-only version, and $399.00 for a package containing both OS/2 and DOS versions.

Randall Howard, *Mortice Kern Systems, 35 King Street North, Waterloo, Ontario, Canada N2J 2W9. Mr. Howard is a founder of MKS and Vice President of Research and Development. He is the principal architect in the development of MKS Toolkit and the MKS POSIX implementation. Mr. Howard has over sixteen years of experience with UNIX systems internals, and more than seven years experience with UNIX standards.*

Software Tools

Applications Manager for PM Development

by Bill Thorne

Applications Manager (AM) from Intelligent Environments offers a rapid, integrated environment for building OS/2 Presentation Manager (PM) line-of-business applications. AM's unified approach enables the application developer to take full advantage of the OS/2 Extended Edition (OS/2 EE) user interface, database, and communications facilities. AM provides an interactive, graphics-based PM development environment for easily creating and maintaining applications.

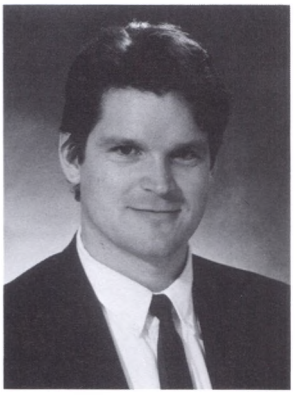
OS/2 EE offers business-systems developers a powerful suite of development interfaces and operational support. A strategic information systems goal for many companies is to use these facilities to integrate PM user interfaces at the desktop level with relational databases and advanced communications to mainframe transactions.

AM AND PM

Not only does Applications Manager develop PM applications, the AM development environment itself is a fully implemented PM application. It offers multiple child windows and a variety of views, program objects, and actions.

The logic entry and editing of an AM program is executed within multiple edit windows, while different levels of the logic are accessed via mouse interaction. Various program operations, such as assigning values to variables and testing results, are prompted using pop-up, context-sensitive dialog boxes, as shown in Figure 1. User entry forms (application dialog boxes) are created and edited within the same environment.

The term *visual programming* describes the AM development environment. Rather than dealing with long program listings (lines of code), the programmer deals with the AM program as a collective group of logic and data abstractions. These program elements or *objects* are created and modified by pointing with the mouse, and then "opening" the command, display form, or other program element by double-clicking. An appropriate PM dialog box then appears that contains PM controls for the attributes of the program element being created or modified. When the element is copied to other portions of the AM program, its attributes are copied as well.



Bill Thorne

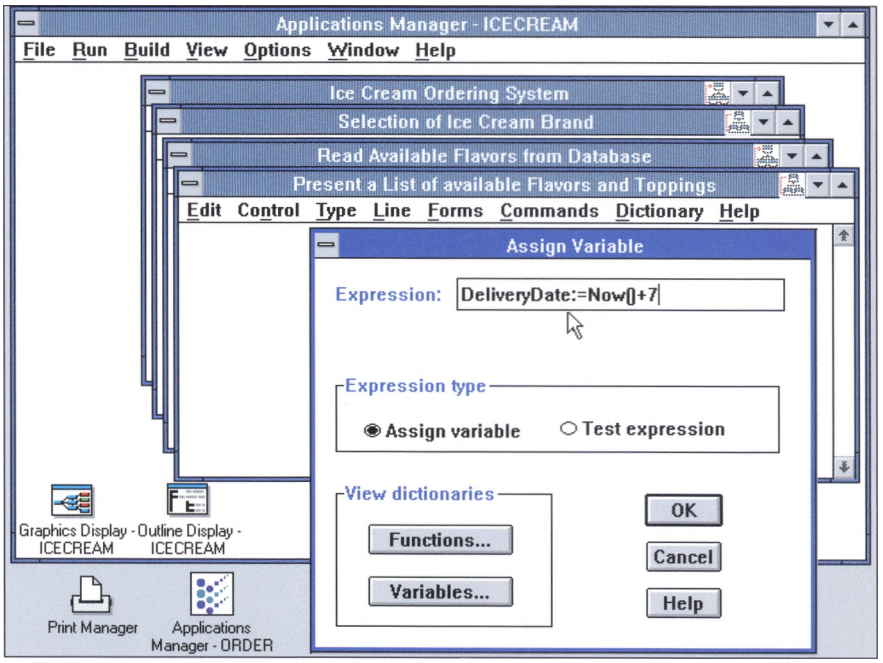


Figure 1. Dialog Boxes in Applications Manager

Developers can build prototypes easily and can run short test cycles.

Visual Programming with AM

The AM-integrated dialog box editor provides unprecedented integration and productivity to PM application developers. Within this editor, PM controls, such as listboxes, radio buttons, and entry fields, are defined using a full-function PM interface.

Along with the definition capabilities provided through mouse click-and-drag techniques, a full set of alignment and duplication features make rapid screen development easy. In addition, any program variables used in the AM dialog box are easily defined or selected from the AM variable dictionary. These variables are automatically available to other AM program statements.

High-Level Logic

AM is based on a tree-oriented, rule-based approach commonly used in expert system development. Developers create this outline tree, or logic flow, with descriptive text using their own words at each step. When tied to the interactive graphical user interface of the AM development environment, this flow offers rapid, simplified entry of complex logic.

The benefits of this approach are that AM programs have few "undocumented anomalies" (bugs), and are easily maintained and extended. The approach also offers author independence; that is, AM programs are self-documenting. Their visual nature also makes them easy to understand and modify.

Interactive Development

When creating graphical user interfaces with AM, developers can build prototypes easily and can run short test cycles using the wide variety of AM's "look and feel" options.

Developers can test run programs directly from an action bar selection on the AM development parent window at any time. Source level tracing is also available for rapid isolation of errors.

When the program finishes the test run, AM automatically displays the development parent window. The developer can immediately make any necessary changes to the program. When tracing the program, the developer can leave execution of the test program and have the statement or display form currently executing displayed in an edit window.

CREATING AN OS/2 PM PROGRAM WITH AM

An AM application is created by the application programmer in a direct fashion without any compiler requirements (such as variable, function, or procedure declarations) or complicated syntax. While AM shares the productivity characteristics of many 4GL's, there is no presumed logic flow. The AM developer has complete flexibility in the design and implementation of the program logic.

Building the Initial Application Logic

The first step in developing an AM application is to open the initial development window using a standard pull-down in the AM development environment (Build Procedures). In a new AM application, the developer can start by describing the general flow of the application in a top-down logic approach. In this stage, each section of the AM program (called a procedure) is described in the developer's own words. The logic flow of the program is controlled simply by linking the high-level procedure descriptions together in a series of AND and OR relationships.

When a procedure contains multiple steps, each of them in turn a lower level procedure, the developer double-clicks on the higher level procedure, opening a new window in which to type the next lower level procedure descriptions. A top-down hierarchical logic tree is developed describing the architecture of the program.

In addition to the overall program logic, additional event-driven program entry points can be attached to menu pull-downs (application action bar), screen controls, or background window processes, as described in later sections.

Once the logic of the program has been described by entering the procedure descriptions, the developer opens procedures by double-clicking on the desired procedure. A variety of command lines can then be entered which perform such actions as displaying PM

dialog boxes, assigning variable values, executing functions, testing conditions, and so on. A procedure line is initially created by using a pull-down selection (or accelerator key).

This method of program design and logic control provides rapid development, enhancement, and maintenance of AM applications. In addition, a variety of integrated tools are included in the AM development system for viewing the logic structure both as a graphical tree and a dynamic outline format (see Figure 2).

Creating Windows or Dialog Boxes

To display a PM dialog box for user interaction, developers attach a Display Form command to a procedure line by using the "Commands - Display Form" pull-down.

By selecting Display Form, the programmer starts the integrated AM Screen Painter (analogous to the toolkit dialog box editor). The dynamic PM window that pops up is a template for the dialog box that the user will see when the application runs. The screen painter includes pull-down menus that control creating a variety of PM controls (entry fields, radio buttons, list boxes, and so on), aligning existing controls, cutting, copying and pasting controls, and other useful actions.

At any time, the developer can double-click on the "frame" of the form being edited, and modify attributes like the border, the title of the dialog box during execution, the default colors, fonts, and so on.

Creating, Editing, and Reusing PM Screen Controls

As the developer designs the display form with the AM screen painter, the Controls pull-down is used to select a type of control. The mouse positions the control at the desired area of the screen. Size is controlled by "dragging" a corner of a sizing frame, and then releasing the mouse button.

Following the selection of a PM control (see Figure 3) and its initial placement on the screen, a dialog box pops up, with which the developer interacts to define the attributes of the control being created. In some cases, literal text may be used as part of the control's attributes. When an AM program variable or expression is more appropriate, the variable name or expression may be entered and integrated with the control's definition. No separate link step is required; all integration between the defined screen variables and the rest of the AM program is automatic and transparent to the AM developer.

Once a control has been defined, its attributes can be modified at any time by resizing, moving, or editing. Because of the integration within the AM development environment, there is no need to modify other parts of the program to reflect the changes in the control.

The AM screen painter fully supports the PM clipboard. Standard clipboard operations enable a single control or group of controls to be marked and cut or copied to the clipboard. The clipboard contents can then be pasted to other portions of the same form, or to a different form. When controls are copied via the clipboard, all attributes are copied also, providing excellent reuse of existing controls and any attached logic.

Attaching AM Logic to a PM Control

Logic is attached to a PM screen control by entering the name of an AM procedure into the control's definition dialog (which is accessed when the control is created, or later by editing an existing control). If the user modifies the state of the control (for example, by entering data into an entry field or clicking on a pushbutton) while the application is executing, the AM procedure attached to the control is executed. This can provide immediate validation or a means to collect additional information.

At any point while the attached procedure is executing, additional dialog boxes (display forms) that are defined will "pop up" over the form in use. Many layers of module pop-ups may be displayed whenever appropriate for the application.

As with other control attributes, both a control and any logic attached as a procedure name are copied when a control is copied from one form to another.

SQL commands may be executed within AM.

Pull-Down Menu (Action Bar) Integration

A main development environment pull-down menu provides the definitions of pull-down menus and main procedure logic to the AM programmer. Two menu selections related to managing the use of pull-down menus are Build Menus and Build Menu Procedures.

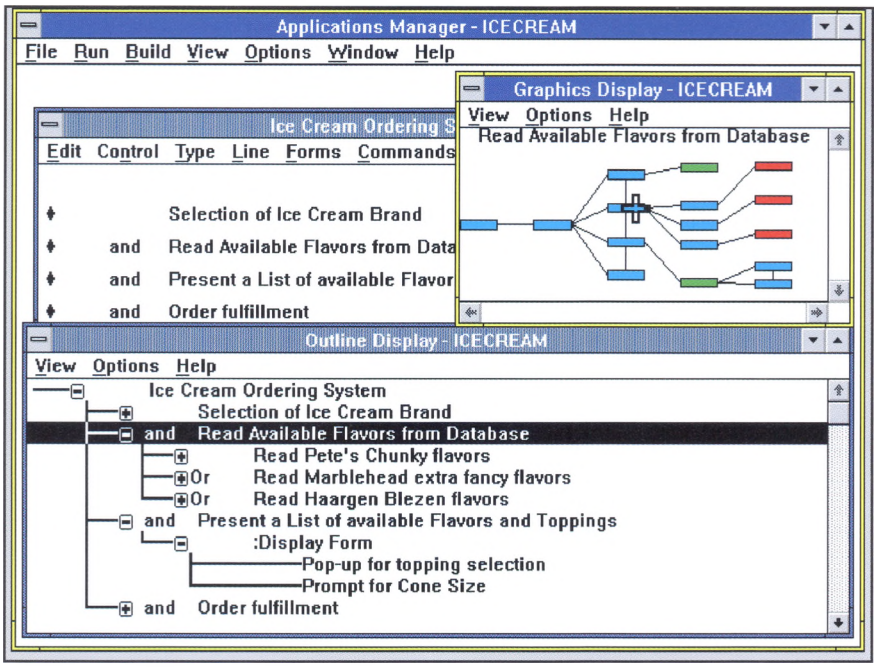

Applications Manager - ICECREAM
File Run Build View Options Window Help
Graphics Display - ICECREAM
View Options Help
Read Available Flavors from Database
Ice Cream Ordering S
Edit Control Type Line Forms Commands
Selection of Ice Cream Brand
and Read Available Flavors from Data
and Present a List of available Flavor
and Order fulfillment
Outline Display - ICECREAM
View Options Help
Ice Cream Ordering System
Selection of Ice Cream Brand
and Read Available Flavors from Database
Read Pete's Chunky flavors
Or Read Marblehead extra fancy flavors
Or Read Haargen Blezen flavors
and Present a List of available Flavors and Toppings
:Display Form
Pop-up for topping selection
Prompt for Cone Size
and Order fulfillment

Figure 2. Logic Structure of Applications Manager

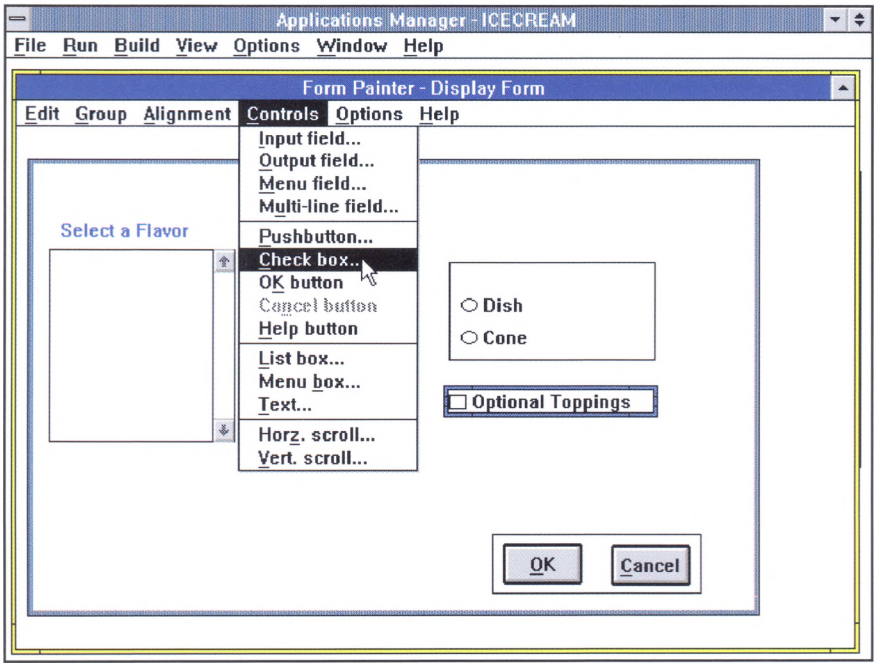


Figure 3. Selecting a PM Control

The first step to integrating an action bar in an AM application is selecting the Build Menus action. This provides an interactive dialog box for the developer to create and edit the text that appears in the runtime action bar area of the PM application. A tilde (~) in front of the key to be used identifies accelerator keys (this is a standard PM convention).

The next step is to select the Build Menu Procedures pull-down, which provides a listbox of the existing menu definitions. It also provides an entry area for the name of the procedure to be executed when the corresponding menu selection is made by the user.

At runtime, if the user makes an action bar selection, any procedure "attached" to the action is immediately executed. Any display form contained in the action procedure will be "popped up" over the current window. When the action procedure is completed, the AM developer can control whether the current form continues to execute, or can end it to branch to another area of the application.

The final step in integrating pull-downs is to control the "graying-out" of selections not available during certain operations within the finished application. An AM function (MenuEnable) is provided to accomplish this.

Accessing and Updating SQL Databases

Among AM's key functions are its Structured Query Language (SQL) database access capabilities. SQL commands may be executed within AM to retrieve selected rows, insert new rows, or update existing rows.

While editing a procedure, the AM developer includes SQL access by selecting the "Commands - SQL" pull-down from within the main development environment. This displays a dialog box that includes a variety of options relating to the SQL request. It also includes a free-form multiline edit field for the programmer to key the SQL statement to be executed at runtime. A pushbutton enables inserting database column names interactively through a database columns

listbox. When the SQL definition dialog box is completed, the SQL command created is attached to the current procedure line, and executes when that portion of the AM program is executed.

For SELECT statements, the full select set (result rows) is returned to an AM *table*. AM tables are special data structures that are very similar in concept to the relational table model. Each AM table has a name that describes the entire table, as well as any number of "columns," which may be AM text or numeric variables. These columns are then organized internally into rows and referenced by the AM programmer by row number and column name.

To integrate an SQL SELECT statement with AM variables, the developer simply defines an AM table's column names with the names corresponding to those of the SQL columns. When the SQL command is executed, AM automatically matches the names of the external data (the SQL columns) with the names in the internal AM table. The programmer can then reference any row or column of the results by using the table name referenced in the SQL command.

Because AM listbox screen controls fully support AM tables (including the display of an entire row in one line of the listbox), the results of an SQL select can be easily viewed by naming the result table in a display form listbox control definition.

Testing and Completing the Application

An AM application can be tested at any time during development by selecting the Run pull-down from the main development action bar. The application under development is executed as if the end user were executing it. There is no need to complete all defined procedures. At runtime these appear to the developer as message boxes. The developer can continue the application by responding to the message affirmatively; no crashes, no burns.

The process of developing graphical user interfaces benefits significantly from AM's interactive nature. The AM developer can design the PM screen and immediately test its "look and feel" during execution. Most AM application testing is actually not testing in the traditional sense (finding bugs), but is a reiterative enhancement of the user interface.

When the application is completed and ready for delivery, the AM delivery system (a separate program) "locks" the module built using the development system. Locking a module prevents any changes or access by anyone other than authorized developers.

Once locked, application modules are distributed to end users, who execute them using the AM runtime support.

INTEGRATING AM APPLICATIONS WITHIN THE ENTERPRISE

Database Integration

The Database Manager SQL support for AM allows advanced development techniques that are uncommon in traditional programming environments. For example, up to four independent SQL tasks can be managed within a single AM module, with each task running on a separate OS/2 thread. This allows selects, updates, inserts, and deletes to operate in the background while the AM program provides the user an entry or information display screen.

Developers can design the PM screen and immediately test its "look and feel" during execution.

Along with support for separate thread SQL execution, AM also provides support for row locking through cursor control, held cursors for maintaining a large fetch between commits, automatic commits, and other advanced SQL features.

Advanced Communications

AM supports OS/2 EE Communications Manager facilities for communicating to both host 3270 transactions using OS/2 EE Emulator High-Level Language API (EHLLAPI) and Advanced Program-to-Program Communications (APPC) transactions using the OS/2 EE APPC support.

To support the broad skill-set of applications programmers, Intelligent Environments supplies special facilities for defining communications characteristics at the administrator

AM supports OS/2 EE Communications Manager.

level. These characteristics include 3270 screen layouts, APPC conversion types, buffer record layouts, and other definitions that are best maintained externally from each AM program.

The AM programmer uses the "name" of a configuration object when issuing high-level commands. For example, a high-level command might be reading an entire 3270 screen into an AM table record, or starting (allocating) an APPC transaction, and then sending and receiving logical records from AM table rows.

A PM-based configuration object editor (Stored Definitions Editor), a set of compilers, and a runtime manager (Stored Definitions Manager) implement this concept. With the stored definitions approach, changes in the communications environment are entered only once, using the Stored Definitions Editor. All dependent AM programs continue to operate without requiring any further modification.

An OS/2 EE SQL database maintains the stored definitions. For this reason, customer-written programs can modify them where initial definition or ongoing maintenance can be leveraged from existing information in the host, such as host CICS screens, mapping layouts, or APPC transactions defined by a CASE environment, such as AD/Cycle.

Finally, any special interfaces or customer-specific calculations that are best performed in C can easily be integrated with AM using the C developer's toolkit. The C toolkit provides a table-driven method for defining new AM functions that can be included transparently in the standard AM function dictionary.

The broad support for all AM data types, and the ability to pass anything from single variables to entire AM tables, make the AM C interface an ideal tool to expand AM well beyond its standard capabilities.

Price and Availability

AM Professional is available now for both OS/2 Versions 1.1 and 1.2, priced at $5,000 per developer. Runtime licenses are provided at a single-unit price of $250.

AM Personal is available also; it includes the same development productivity as AM Professional without the extensive database and communications support. AM Personal is available for $695 per developer. It is appropriate for personal applications and prototyping.

Bill Thorne, *Intelligent Environments, 2 Highwood Drive, Tewksbury, MA 01876. (800) 669-2797. Mr. Thorne is the U.S. Product Director for Applications Manager. He previously worked for IBM for twelve years. While at IBM, he worked with customers in the insurance, transportation, finance, manufacturing, distribution, health, government, and recreational industries (to name a few). As a developer, he designed and wrote a PC-based terminal emulator currently being marketed by IBM, a 60-node Series/1 network control program, and a travel agency billing system. He holds a BS in Business Administration from the University of Vermont.*

Software Tools

Powerful Voice Processing Under OS/2

by Robert Panoff

Developers are increasingly using new digital communications technologies to integrate voice and data processing and to produce applications offering more effective access to information. Most telephone users have probably already encountered a computerized voice processing system. Natural MicroSystems' Voice Branch Exchange (VBX) is an advanced voice applications development environment with powerful enabling tools. This article describes its use with the extensive real-time multitasking capabilities in OS/2. Together, VBX and OS/2 functions provide an ideal platform to help developers use voice processing to provide solutions across a wide range of industries.

FAVORABLE END USER ECONOMICS

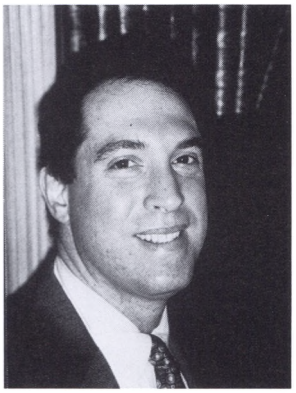
Basic market trends and user economics generate demand for voice processing applications. Voice processing systems perform routine, repetitive, and labor-intensive functions. They reduce users' information distribution costs as well as the personnel problems associated with boring or mundane tasks. Voice processing developers have, at their disposal, 120 million computer terminals, in the form of telephones in the public network. Together, these factors translate into major opportunities for voice processing developers. VBX's advanced tools and OS/2's high-function operating environment enable developers to deliver quickly on low-cost voice processing applications with intrinsic value to their users. The rapid growth of voice processing systems (over 50% per year, from $500 million in 1988 to an expected $2.6 billion in 1992) points to the outstanding opportunities available in virtually every vertical industry.

VOICE PROCESSING SOLUTIONS USING OS/2

The time and effort required to develop and maintain an application play key roles in determining its ultimate success. To minimize development effort, VBX offers a complete set of integrated function libraries, editors, utilities, and debuggers in a modular, open architecture. This architecture relies heavily on OS/2's structure and function.

The VBX product architecture, shown in Figure 1, is divided into seven levels, starting with the computer platform, and ending with the application itself. From the developer's point of view, VBX hardware and software may be treated independently. The number or type of VBX voice adapter circuit boards may be changed by installing or removing them from the system. They are independent of the VBX software supporting them. VBX software includes a hardware driver level, a function library level called Multiline Engine/2 (ME/2), and VScript, a high-level scripting language designed to accelerate application development. Applications may be written at any level, and can easily communicate or be integrated in a system.

VBX's multiline OS/2 driver supports voice adapters on the AT (one and four channel adapters) bus, and Micro Channel (one and two channel adapters) architecture. Each VBX adapter has significant on-board processing power, a multitasking operating system (VMOS) that manages a dedicated Digital Signal Processor (DSP), and 16 KB of RAM per voice channel. Using this processing power, board-level software produces a quality voice and excellent telephone call supervision, without the Touch-tone® detection problems experienced by most voice process-



Robert Panoff

ing systems. While information is provided to help developers write DSP microcode for special requirements, most OS/2 programmers never need these capabilities. Communication between the voice adapters and the driver is at a low level; a byte stream of commands is sent to the adapter, and a byte stream of events returns. At the driver, each event causes an interrupt, and one special event triggers the transfer of a frame of data in either direction (or both for transcoding). At the lowest level, frames are split into blocks for best performance. The driver performs additional buffering to support the OS/2 generic IOCTL interface. Frames may be grouped into multiframe buffers that are queued going to or from the voice adapter.

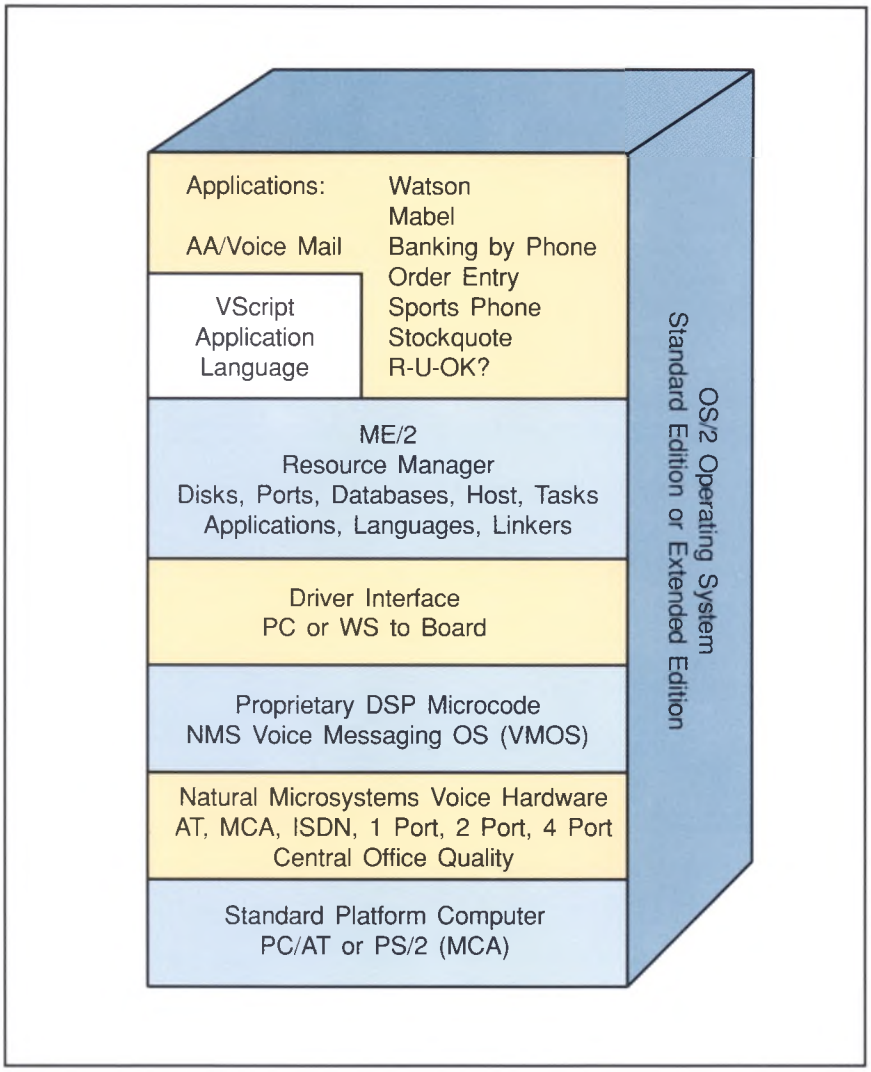


Figure 1. VBX Product Architecture

ME/2 includes a set of Dynamic Link Libraries (DLLs) to support runtime operations, a development editor, a system supervisor, and several support utilities. Although minimal functions are required to begin implementing simple applications, an extensive library (135 speech and telephone management functions) is callable from any application. An application either controls calls or monitors the activity on all telephone lines without exerting control over them. ME/2 is highly parameterized; developers modify default values of more than 170 variables to regulate the processing of their applications. All ME/2 line access is controlled by standard OS/2 device Open, Close, and IOCTL calls. With application development, the ME/2's managers deal with a single line whenever possible. OS/2's cloning of DLL functions extends the application functionality to the desired number of telephone lines. Application programmers can concentrate on program function without concern for the complexities of multiline resource allocation, scheduling, prioritization, and arbitration, which are necessary in non-OS/2-based voice systems.

VScript is a high-level, English-like command language that makes the capabilities of ME/2 available to a larger group of developers. Using VScript, developers shorten development time and reduce the maintenance effort for the life of their application. VScript, built upon the ME/2 kernel libraries, includes the VBX Supervisor, a VScript Development Editor, a VScript compiler, and a VScript runtime interpreter. All application-callable ME/2 functions are available through VScript. In addition to ME/2's extensive voice and telephone functions, the VScript language contains commands for basic arithmetic, string manipulation, and flow of control operations. A VScript program consists of a single high-level command per line in standard ASCII text, accessible from any text editor. When functions beyond the VScript language are required, externally developed modules or OS/2 Application Programming Interface (API) functions can be called. The developer can incorporate these new functions into VScript by defining a new VScript command corresponding to the new function developed. This permits the complexity of call management to be contained within VScript, while permitting great flexibility in perform-

ing special computations or in accessing host computers or PBXs.

VBX USES OS/2 STRUCTURES AND FUNCTIONS

The VBX kernel executes many system management functions using OS/2's DLL structure and multithreading. This minimizes the code that programmers must build and maintain in their applications. Kernel DLLs are grouped together into a number of semi-autonomous ME/2 managers related by their functionality. DLLs for ME/2 are built in accordance with OS/2 API standards. Consequently, VBX applications can be written in any programming language supported by OS/2's Dynamic Linker. Any application module can be easily linked into the overall system using these conventions.

Single-function calls to ME/2 managers are all that is necessary to perform operations such as dialing phone numbers, recording (or playing) speech to (from) a disk file, and collecting or decoding Touch-tones. The interface to each manager is sufficiently defined to allow developers to replace individual managers if they wish to support special functionality (such as different databases).

The DLL structure is well-suited to the complex resource management tasks found in multiline voice processing systems. Such systems manage many common system resources at the same time. Each application requires many telephone lines, voice channels, speech files, and data files, while also processing real-time telephony events. Within VBX, the management of common resources, like telephone lines or voice files, is concentrated in a DLL designed specifically for that task. Since DLLs are global routines, they are shared by VBX applications running in many different OS/2 sessions. DLLs also control their own dedicated data block used to track a large number of common resources independent of any individual application. VBX's use of DLLs frees the application and the programmer from these system management tasks.

DLLs significantly affect voice processing performance. If many voice channels simultaneously share DLL code, frequently called DLLs often remain in system memory. Reducing the number of code segments swapped in and out of memory reduces memory requirements and improves overall system performance.

Voice processing systems must always be ready to process many real-time events—ring and answer detection, beginning (ending) of speech file processing, and generating (detecting) Touch-tones, to name a few. With OS/2's multithreading capabilities, up to 255 asynchronous subprocesses may be automatically launched to track and to respond to a large number of real-time events. Therefore, an application must create and supervise many subroutines throughout the application to accomplish this. Using OS/2 in managing real-time events frees the programmer to concentrate on higher level functions.

Natural MicroSystems encourages VBX application programmers to use DLLs and threads to improve application-specific resource management and performance. The maintenance and installation of new software versions are easier when applications are developed as DLLs. If only a few modules have been changed, there is no need for the entire application to be relinked and recompiled. Instead, DLLs reduce installation time by allowing the developer to distribute new versions of only those modules containing changed object code. After installation, OS/2 dynamically links the new DLLs into the applications that require them at runtime.

Using OS/2 in managing real-time events frees the programmer to concentrate on higher level functions.

EASY HOST ACCESS THROUGH COMMUNICATIONS MANAGER

Many interactive voice-response applications require host database access. Traditionally, such systems require extensive programming to accommodate the specific hardware configuration used. The DLL structure of the OS/2 Communications Manager offers easy access from any application without any special coding for VScript or ME/2 programmers. Extensive Communications Manager functionality permits the developer to select the communications protocol and hardware device most appropriate for the application.

Voice-response applications typically provide telephone access to existing host databases. The VBX system maintains control of the voice application. By using 3270 terminal

Tools for one-pass development include speech, database and parameter editors, program debuggers, and online help.

emulation, the system often eliminates the need for additional host system programming. The VBX application can directly call extensive Emulator High-Level Language API (EHLLAPI) functions to interact with virtual I/O screens for each session needing host access. EHLLAPI's high-level functions make it easy to copy a Touch-tone string into a field for transmission to the host, or conversely, to search a field for data which the VBX application outputs over the telephone line. Many voice-response applications share EHLLAPI functions. OS/2 Extended Edition (OS/2 EE) 1.2 supports up to thirty-two concurrent sessions providing excellent host access capacity.

APPLICATIONS IN FRACTION OF TIME

Developers build VBX applications using a highly interactive, integrated development environment. Easy access to tools for one-pass development, including speech, database and parameter editors, program debuggers, and complete online help, revolutionizes the programmer's approach to voice processing systems development.

VBX developers produce their applications in a series of usable modules, debugged with speech and data included, as the program modules are completed. Figure 2 contains a portion of a simple VScript telephone attendant application with three labeled subroutines (ANSWER, ANSWER_a, and GET_TT). This VScript application answers an incoming call on the first ring, and solicits Touch-tones from the caller indicating the desired telephone extension, before performing other subroutines to actually transfer the call. It also includes one speech file (AutoAns) with two independently addressable speech segments (1 and 3). To record speech when AutoAns is first referenced, the programmer presses the F5 key to get the Speech Editor on the monitor. The Speech Editor's menu offers the developer Record, Play, Stop, Help, and Compression options. When the developer selects Record, a prompt appears for the speech file name and the speech segment number. The developer also is instructed to pick up the receiver of the local telephone. Speech is recorded immediately after a tone is heard in the telephone ear piece, and continues until stopped from the keyboard or when the telephone receiver is placed back on-hook. The speech can then be passed onto a voice editor that can cut, trim, and paste speech files with audio quality (see Figure 3). Therefore, all speech required to fully test application modules is quickly and easily recorded without exiting the VBX development editor.

OS/2 enhances the productivity of the VBX development environment. Proficient devel-

```
!ANSWER
     waitfor( INCOMING_CALL, 0 )
     localmute( 1 )
     Flg   = 0                      ;  Reset initial database to be used.
     Pass = 0                       ;  Reset retry counter.
     flushtt                        ;  Flush touchtone buffer.
     playlist( "AutoAns", "1,3" )   ;  Play greeting extension request.
     goto GET_TT                    ;  Branch accordingly.
  !ANSWER_a
     playmsg( "AutoAns", 3 )        ;  Play extension request message.
     goto GET_TT                    ;  Branch accordingly.

!GET_TT
     clear( extension );            ;  Clear buffer.
     extension = rectt( Ext_Limit );  Get the extension from the caller.
     extension = VM_Ext?   pass += 1    goto VOICEMAIL;Check for next
     extension > 9?        pass += 1    goto TXFER    ;branching based on
     extension = 0?        goto OPERATOR              ;touchtones entered,
     stopped?              goto CHECKPASS;            ;or not entered.
     pass += 1;                                       ;
     goto OPERATOR                                    ;
```

Figure 2. Portion of VScript Automatic Telephone Attendant Application

opers can work on multiple related or unrelated VBX applications at the same time, or run a VBX application in one OS/2 session, develop code in a second session, and debug in a third.

The developer can quickly assemble different working modules into an extensive application, which accelerates the completion of the finished product. Using the VBX development environment and its tools, developers can produce applications in 80% less time than was previously required with non-OS/2-based systems.

SUPERIOR PERFORMANCE AND EFFECTIVE TRANSACTION PROCESSING

Real-time events require a response. Lines ring, Touch-tones are received, and recording speech files all begin and end in real-time. These events can occur simultaneously on 24, 32, or even 40 telephone lines. With DOS, developers must produce quantities of application code to handle these multiple tasks and their priorities, to allocate resources and arbitrate among the competing tasks, and to constantly poll those resources to determine readiness for the next task. As a real-time multitasking operating system, OS/2 helps VBX produce outstanding voice processing performance but does not require this excess code. With OS/2 the developer can use several task or process levels to track and respond to real-time events. Each process is assigned a priority. OS/2 automatically manages the priorities and resource allocation accordingly. Real-time processes can be assigned preemptive priorities to ensure their availability for real-time events. When an event occurs, OS/2 responds. When nothing is required of OS/2, it suspends processes so that no processor time is lost. Applications need not poll system resources; therefore, VBX's performance in voice-driven transaction processing is significantly improved.

OS/2 POTENTIAL

The four applications described below illustrate the advantages offered by an OS/2-based voice processing system. These applications were selected from among many offered by over one hundred Natural MicroSystems developers currently working with OS/2 and VBX.

A Georgia-based Value-Added Reseller (VAR) had been offering their automated attendant/voice mail systems on a national basis. To accommodate their customers' special requirements and meet their objectives for expansion, they needed to build systems with more reliable telephone call supervision, higher line capacity, and greater flexibility. VScript enabled them to duplicate the functionality of their original system in 20% of the time required by available non-OS/2 systems. OS/2's multitasking and dynamically linked structures provide an opportunity for this VAR's dealers to independently add a number of additional application modules to meet the special needs of their customers.

A developer has produced a system for a large Chicago Board of Trade options brokerage firm that provides commodity quotations via telephone to thousands of the firm's clients. Using OS/2's real-time multitasking capabilities, this system simultaneously provides options information to callers on thirty-two telephone lines, while it receives a real-time data feed from the market over a high-speed communications link. This system uses a 25 MHz 80386-based system that holds eight VBX400 four-port voice boards. The developer is currently enhancing the ME/2-based system, using OS/2's ability to launch and suspend multiple threads, and intends to extend the performance to forty simultaneous telephone lines.

A New York City pension and benefits accounting firm implemented an interactive voice response system for its clients. The clients can put many different companies' pension and benefit accounts on the system, allowing each company's employees to inquire about the status and value of their benefits. Only a high-performance multitasking operating system like OS/2 could man-

age many different benefits databases, while also providing secure access to callers over a variety of telephone lines. The system was so well received that the developer has built a high-level, fill-in-the-blanks application generator to help clients add their own voice response applications after only a few hours of work.

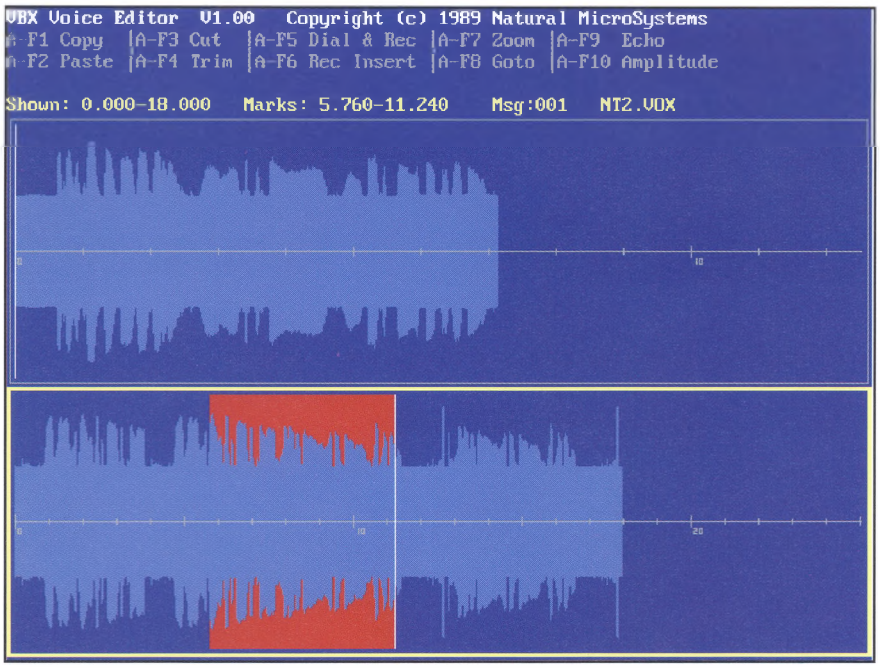

VBX Voice Editor V1.00 Copyright (c) 1989 Natural MicroSystems
A-F1 Copy |A-F3 Cut |A-F5 Dial & Rec |A-F7 Zoom |A-F9 Echo
A-F2 Paste |A-F4 Trim |A-F6 Rec Insert |A-F8 Goto |A-F10 Amplitude
Shown: 0.000-18.000 Marks: 5.760-11.240 Msg:001 NT2.VOX

Figure 3. Voice Editor with Speech Segment Marked for Editing

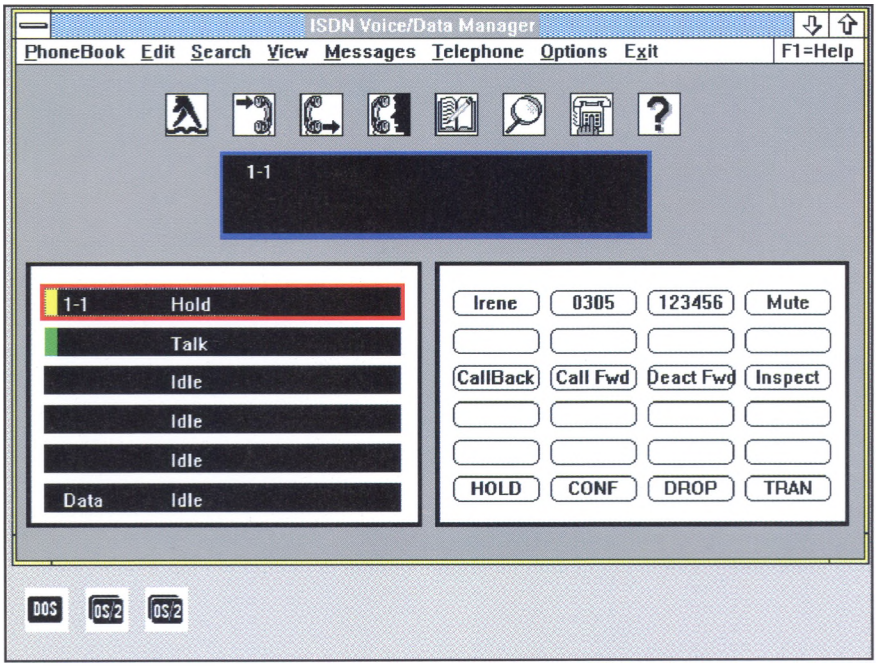


Figure 4. Presentation Manager Interface for ISDN Voice/Data Manager

NCR used OS/2 and the VBX architecture to produce its new Integrated Services Digital Network (ISDN) product. *PC Magazine* (February 13, 1990) declared the NCR ISDN Voice/Data Manager as its "Editor's Choice" among the six ISDN PC terminal adapters it reviewed. *PC Magazine* said, "OS/2 programmers can use standard read/write calls to pass data over the NCR ISDN adapter. This is an intelligent move because with ISDN and OS/2 both inherently multitasking, they make an excellent match." The Voice/Data Manager uses Presentation Manager (see Figure 4) to give the PC user "point and click" control over concurrent voice messaging and ISDN feature phone functionality.

OS/2 FUTURE IN VOICE PROCESSING

Voice processing solutions are widely applicable across a broad spectrum of organizations. They have the potential to provide better information, improved customer service, and lower costs. Since voice processing systems increasingly involve multiple applications and access to host databases, OS/2's real-time multitasking capabilities give developers a decided advantage in getting their applications to market quickly.

Robert Panoff, *Natural MicroSystems, 8 Erie Drive, Natick, MA 01760. (508) 650-1300. Mr. Panoff, vice president, new business development, focuses on the development of strategic and key customer relationships. He has held a variety of sales and marketing positions at Natural MicroSystems, Digital Equipment Corporation, and Unisys.*

International

OS/2: A Worldwide Product

by Cheryl Bullen

English OS/2 1.2 Standard Edition is available in Europe with language versions becoming available during Spring 1990. OS/2 Extended Edition 1.2 (English and national language versions) will be available in the E/ME/A (Europe, Middle East, and Africa) region during 2Q 1990.

Germany

The German version of the *OS/2 Application Guide* contains 244 applications including 20 for Presentation Manager. Order number is IBM GF12-1656-1.

The CeBIT Computer Fair in Hanover was held on March 21–28. Many of the demonstrations were developed using OS/2. These will be updated and shown at the Orgatechnik Computer Fair in Cologne in October.

ENFIN/2, a 4GL object-oriented development environment tool, was announced in Germany by ENFIN Software Corporation and demonstrated at the Hanover Fair.

Switzerland

Contact your IBM Representative for details of upcoming OS/2 events.

Italy

An IBM-sponsored "Software Market" was held in March for dealers, software vendors and agents, who presented their application solutions for both the PS/2 and AS/400. IBM had a large OS/2 location with its business partners in Italy, showing both external software applications and IBM solutions.

The conference will be held in the "Fiera Di Milano."

E/ME/A

Two OS/2 1.2 seminars in Vienna last September generated such a large demand by developers that the seminar will be repeated during the last week of May in Malaga, Spain (May 28–June 1). For more information, contact the IBM OS/2 representative for your country.

Cheryl Bullen

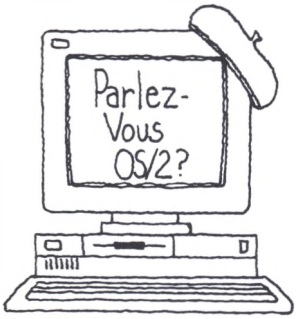
Cheryl Bullen, *IBM WS/SPM, Mountbatten House, Basing View, Basingstoke, Hampshire RG21 1EJ, England. Telephone: 44-0-256-475050 X8049. FAX: 44-0-256-58684.* *Ms. Bullen is an OS/2 product marketing manager in the E/ME/A OS/2 Project Office. She joined IBM in Boca Raton, Florida in 1983 working in the newly formed Entry Systems Division. She went to England on international assignment as an E/ME/A OS/2 SE product manager in 1987. Prior to joining IBM, Ms. Bullen held positions as a university director and dean. Ms. Bullen holds a BA in Journalism and English from the University of Minnesota/Minneapolis, an MA in Higher Education, and has completed all but the dissertation for her DBA.*

International

OS/2 National Language Support

Nicola Colbourne

Vladimir Radnovic



by Nicola Colbourne and Vladimir Radnovic

The OS/2 system has been translated into eleven languages, serving nineteen different countries. These translated versions sell as well as the U.S. version, indicating OS/2's popularity in Europe. OS/2 developers can write applications that work in the U.S. as well as in countries with different alphabets, keyboards, and currencies. OS/2's built-in National Language Support (NLS) simplifies international installations for both the programmer and the user. With a little additional planning and coding up front, an application can be marketed worldwide.

The first IBM PCs quickly became popular outside of the U.S., but their users faced an immediate drawback: these computers "spoke" only English. Not only did DOS commands and messages appear in English, but the U.S. keyboard design was unfamiliar to non-U.S. users. Although the PC had a built-in native set of 256 ROM characters, including accented vowels, currency symbols, and other non-U.S. characters, they were difficult to use. And to make matters worse, many of the early PC software applications were written with only U.S. users in mind.

Soon, country-specific keyboards were sold with PCs in Europe. To accommodate national keyboard mapping to the PC's native character set, national code pages, or character subsets, were designed. A text file created with the French code page, however, might not look the same when viewed on an Italian PC. This caused problems when exchanging files between countries, so a standard multilingual code page was created. Now, both DOS and OS/2 use code page 850 in Europe for English, French, German, Italian, Spanish, Portuguese, Dutch, Swedish, Danish, Finnish, and Norwegian. Today, a PC user in Germany can easily configure the operating system for the German alphabet, keyboard, and other national conventions without sacrificing data compatibility.

TRANSLATED OS/2

The translated versions of OS/2 available today are listed in Figure 1, along with other country-specific options. The matrix lists nineteen countries, eleven languages, and seventeen keyboards. Some countries share a language, keyboard, or both. For example, France, Switzerland (French-speaking), Belgium, and Canada use the French keyboard. Some countries, like Switzerland, use multiple languages and keyboards, requiring an easy transition from one code page or keyboard to another. In fact, in countries such as Switzerland and Canada, some customers use multiple languages within the same company, and sometimes on the same system. (See "Country-Specific Options" below for description of Yes/No abbreviations.)

Point and Click

Operating system commands, such as DIR and PRINT, are still in English today, as accommodating duplicate command sets (and reserved words) for multiple languages would require much additional overhead. Fortunately, the popularity of graphics interfaces has lessened the dependency on operating system commands. (If a picture is worth a thousand words, then an icon is worth a thousand commands.) In most cases, the Presentation Manager (PM) user interface has been translated, which isolates the user from OS/2 commands.

Most operating system messages, such as "press any key to continue" have been translated. Some software tools, compilers in particular, may insert their own English message (such as "Stack Overflow") into application code. Fortunately, the user rarely sees most of these messages.

Translated Applications

Once a user starts an OS/2 application, the interface should appear in the local language. Ideally, users should be totally unaware that the application was written in one language and translated later. The relative ease or difficulty of the translation process depends on the original application design, especially on its use of external text files. IBM's Personal Editor II is a good example of an application with user-modifiable external text files; therefore, it is easily translatable.

CODE PAGES

Both OS/2 and DOS support multiple code pages. Each code page contains common letters, numbers, symbols, and other characters common to a particular country or language group. The main code pages that OS/2 supports are:

437 U.S. English (ROM-based)
850 Multilingual
860 Portuguese
863 Canadian French
865 Nordic (ROM-based, in Scandinavian countries)

The most common code page is code page 850, which is used in IBM's Europe/Middle East/Africa (E/ME/A) region. In addition, PM supports the following country-specific code pages:

037 U.S. English
285 U.K. English
297 French
277 Danish/Norwegian
273 Austrian/German
278 Finnish/Swedish
274 Belgian
282 Portuguese
280 Italian
500 International
284 Spanish

In OS/2, the program or user can change the active code page. Two pages may be available simultaneously. One must be code page 850, the multilingual code page, in addition to the ROM-resident code page (code page 437, except in Scandinavia). These code pages are loaded during system initialization, and switching can take place between them. Code pages may be set for each OS/2 session.

Country	FR	SF	BE	CA	GR	SG	SP	LA	PO	IT
Language	Fre	Fre	Fre	Fre	Ger	Ger	Spa	Spa	Por	Ita
Country Code	033	041	032	002	049	041	034	003	351	039
Keyboard	FR	SF	BE	CF	GR	SG	SP	LA	PO	IT
National Code Page	437	437	437	863	437	437	437	437	860	437
Multilingual Code Page	850	850	850	850	850	850	850	850	850	850
"Yes" Abbreviation	O	O	O	O	J	J	S	S	S	S
"No" Abbreviation	N	N	N	N	N	N	N	N	N	N

Country	DK	NL	NO	SV	SU	US	UK	AU	AS
Language	Dan	Dut	Nor	Swe	Fin	Eng	Eng	Eng	Eng
Country Code	045	031	047	046	358	001	044	061	099
Keyboard	DK	NL	NO	SV	SU	US	UK	US	US
National Code Page	865	437	865	437	437	437	437	437	437
Multilingual Code Page	850	850	850	850	850	850	850	850	850
"Yes" Abbreviation	J	J	J	J	K	Y	Y	Y	Y
"No" Abbreviation	N	N	N	N	E	N	N	N	N

Legend:

Dan = Danish
Dut = Dutch
Eng = English
Fin = Finnish
Fre = French
Ger = German
Ita = Italian
Nor = Norwegian
Por = Portuguese
Spa = Spanish
Swe = Swedish

AS Asia (English)
AU Australia
BE Belgium
CA Canada
DK Denmark
FR France
GR Germany
IT Italy
LA Latin America
NL Netherlands
NO Norway

PO Portugal
SF Switzerland (French)
SG Switzerland (German)
SP Spain
SU Finland
SV Sweden
UK United Kingdom
US United States

Figure 1. Translation Matrix

Code pages for the keyboard, display, and printer can be set independently. Code page switching, however, can only take place in printers and displays that support downloadable fonts, including the following products:

IBM Proprinters
IBM Quietwriter III
IBM Enhanced Graphics Adapter and Display (PC/AT)
IBM Video Graphics Array (VGA) support on Personal System/2 (PS/2)
IBM PS/2 Display Adapter 8514/A (PS/2 8514 display)

The OS/2 PM Application Programming Interface (API) also includes calls for querying and setting graphics code pages for Presentation Manager windows. Function calls can convert characters, or strings of characters, from one code page to another. This allows one PM application to share data with another, using a different code page. It is also possible to create a unique font and substitute it in a code page for the duration of an application.

The initial system code pages are set during power on from the "CODEPAGE = " statement in the CONFIG.SYS file. Printer code page switching can be done if the spooler is installed and a "DEVINFO = " statement has been specified for the printer in the CONFIG.SYS file. Device code pages may be changed as necessary to accommodate a specific video display or printed report. This can be done by OS/2 program function calls or from the user CHange Code Page command (CHCP 437, CHCP 850, etc.).

```
.
.
.
line 18   COUNTRY=044,C:\OS2\SYSTEM\COUNTRY.SYS
line 19   DEVINFO=KBD,UK,C:\OS2\KEYBOARD.DCP
line 20   CODEPAGE=437,850
line 21   DEVINFO=PRN,4201,C:\OS2\4201.DCP,ROM=437
.
.
.
```

Figure 2. Example of CONFIG.SYS for United Kingdom

To use country specific options, the CONFIG.SYS file requires some additional changes. In Figure 2, line 18 defines the country code (044) for the United Kingdom. On line 19 the U.K. keyboard is selected. The primary (437) and secondary (850) code pages are defined on line 20, and a Proprinter with code page 437 is defined on the following line. When the user starts the system, OS/2 enables U.K.-specific formats, such as the local time and date conventions. These options are independent of the keyboard selection.

INSTALLATION PROCEDURES FOR NLS

Users may customize OS/2 for use in a particular country in two ways. If a translated version of OS/2 is being used, its installation procedure should be followed, selecting the country at the "Select Country and Keyboard" menu, and specifying a supported printer. After installation, the CONFIG.SYS file should be checked and edited, if necessary, according to the Translation Matrix (Figure 1). Then restart the system. Voilà! OS/2 now speaks the local language.

For the U.S. English version of OS/2, the desired country can still be selected and the country-specific options will be loaded. However, all text will still be in English and the U.S. keyboard will still look the same. For example, a German visitor to the U.S. could use his host's machine by either editing CONFIG.SYS and restarting the system (a more permanent change), or by using the temporary KEYBoard command from an OS/2 window. When KEYB GR is entered, the U.S. keyboard responds as if it were a German keyboard, and the German visitor can "touch type" as if in Germany.

While code page switching offers advantages to software developers seeking foreign markets, it also requires some special considerations. Frequent page switching should be used sparingly since it can affect performance. Only displays, printers, and plotters with a font download capability can support code page switching. It may be desirable to ensure that the application can still run, even with a restricted character set, on a system

without such devices. The application may have to verify that the code pages for the active data file, output device, and the program itself all agree. If this is not done, problems such as the following could occur:

- A data file is created while one code page is in effect. It is then accessed later by another program using a different code page. Some of the characters appear different when displayed or printed.
- A program or a data file is stored with a name that is entered while one code page is in effect. Later, the user attempts to enter the same name, but a different code page is active. The name is not recognized, so the file cannot be located.

A developer may wish to upgrade an existing application to include code page switching. Although it is more difficult to add code page switching to an existing program, proper planning can minimize potential problems.

COUNTRY-SPECIFIC OPTIONS

Country-specific options, like code pages, are different in each country. Country-specific options are:

- Country Symbol
- Country Language
- Country Code
- Country Keyboard
- National Code Page
- "Yes" and "No" Abbreviations
- Other options (Returned by the international call)
  - Time and date formats
  - Decimal separator (number punctuation)
  - Character case map
  - Collating sequence
  - Double-byte character set environment vector

The country symbol is the two-letter abbreviation of a country name listed in Figure 1. Country codes are three-digit numbers in which leading zeroes must be used. In most cases, keyboard abbreviations are the same as the country symbols.

"Yes" or "No" responses, frequently required by OS/2 and its applications, are simplified to single-letter responses, as shown in Figure 1.

The International APIs

Figure 3 shows NLS APIs. The International Call allows a program to interrogate the system to determine which options are in effect. Country information is returned by the use of three API function calls:

- **DosGetCtryInfo** returns the date and time format, currency conventions, and decimal punctuation. Different decimal conventions, for example, are used in these countries:

  1.234.567,89 (Italy)
  1,234,567.89 (Switzerland)
  1 234 567,89 (France)

- **DosCaseMap** indicates the upper/lowercase mapping sequence currently in effect. This takes the form of a 256-byte table with some ASCII codes (such as lowercase letters) "mapped" or "folded," for sorting purposes, as equivalent to other codes. For example, the following list of names will not sort correctly if the native ASCII sequence is used. The lowercase "d" must be mapped as equivalent to uppercase "D." Mapping should not be confused with collating, which determines the alphabetical sequence, or sorting order.

ASCII Sequence	Correct Sequence
Defoe	de Souza
Richards	Defoe
de Souza	Richards

The collating sequence (DosGetCollate) determines the order of all characters, including the mapped ones, in an alphabetical sort. Because of language differences, the desired collating sequence for an alphabet may not correspond to a straight

General Country Information	
DosGetCtryInfo	Get country information table
DosCaseMap	Case map (a binary string)
DosGetCollate	Get the collating sequence table
DosGetDBCSEv	Get DBCS environment vector
Code Page Switching	
DosGetCp	Get the calling process code page and system code page
DosSetCp	Set the code page for the calling process and the session's display and keyboard code page
VioGetCp	Get video subsystem code page
VioSetCp	Set video subsystem code page
KbdGetCp	Get keyboard subsystem code page
KbdSetCp	Set keyboard subsystem code page
Presentation Manager Additional Calls	
GpiQueryCp	Query default GPI code page
GpiSetCp	Set default GPI code page
GpiCreateLogFont	Create a font in a code page
WinQueryCpList	Get a list of available code pages
WinQueryCp	Query code page for translating incoming messages
WinSetCp	Set the code page
WinCpTranslateChar	Translate a character from one code page to another
WinCpTranslateString	Translates a string from one code page to another

Figure 3. National Language Support API Functions - OS/2

The ideal way to deal with text translation is to isolate the text from the executable code into a separate ASCII file.

ASCII sequence. The table assigns a weight value to each ASCII character, arranged according to country convention. These weights, not the ASCII order, control sorting.

- **DosGetDBCSEv** returns the Double-Byte Character Set (DBCS) environment vector, consisting of a mixture of single- and double-byte characters. If a character lies outside the range of eight-bit binary values from 00H to FFH, then it is a single byte. If it lies within this range, it represents the first byte of a double-byte character, usually used to represent Chinese or Japanese characters.

TEXT TRANSLATION

Text translation is an important consideration for the international software developer. Program coding may not change, but messages and other text must be translated into the national language. Often the translator is not the original programmer, and may not be a programmer at all.

The ideal way to deal with text translation is to isolate the text, such as messages, help text, and so on, from the executable code into a separate ASCII file. This allows straightforward translation, as long as the relative positioning of the text within the file remains the same. If a program has to access a message file frequently, performance can be affected. For this reason, many DOS or OS/2 full-screen applications write text at the end of the program itself.

OS/2 PM APPLICATIONS

OS/2's dialog box editor has greatly improved the text translation process by creating text windows that can be modified easily. With or without this editor, developers can facilitate text translation.

- Icons and other graphic symbols are a useful idea in international programs, since they reduce the need for text. However, an icon may not have the same meaning in all countries, so only commonly accepted international symbols, or custom designs for specific countries, should be used.

- In OS/2 PM, national language dependencies can be defined in resource files separate from the program. For example, any strings that consist solely of text should be held in string tables, dialog boxes placed in dialog templates, and so on. These can be translated without changing the application itself.

- The international application developer can also exploit the way OS/2 PM applications link to the resource files. A separate dynamic link library can be created for each language, selected by the program (or user) at either installation or runtime. This would require the programmer to package multiple resource libraries with the application.

- Providing enough space for text translation is important, especially in English-language programs, since it may take up to 50% more space to say the same thing in a language such as German. Any text string stored in a data array is limited by the size of the array, so it is important to make sure the array is large enough to hold translations. The display areas of the PM window title bar and pull-down menus will expand to fit the text to be included. The dialog box editor makes it easy to see the impact of translated text on a dialog box and on the windows with which it shares a screen. Items within the box may be edited, with the results immediately displayed on the screen. If necessary, other adjustments, such as dialog box size or spacing, can be made.

An alternative method of editing is to use an ordinary text editor on the dialog template, which is an ASCII resource file created by the dialog box editor. This file also defines window sizes and positions. A language translator might not have the knowledge necessary to change sizes or space correctly, but could translate the text held there, if enough space was allocated.

SUMMARY

OS/2 is multilingual: its applications can be, too. In at least eleven languages and nineteen countries, OS/2 offers a solid platform for international application development. For the application developer, a little preparation can make text translation and other country-specific modifications much easier.

Nicola Colbourne, *IBM, Mountbatten House, Basing View, Basingstoke, Hampshire RG21 1EJ, England. Miss Colbourne is a systems engineer in the System Software Sales Group. She joined IBM in 1988 from the University of Surrey, where she obtained a BSc in Mathematics and Computing Science.*

Vladimir Radnovic, *IBM Entry Systems Division, 1000 NW 51st Street, Boca Raton, FL 33429. Mr. Radnovic has been with IBM for seven years and is a senior associate programmer in the OS/2 System Test Group. His emphasis is the National Language Support. He has a BS in Computer Science from Florida International University and an MS in Organic Chemistry from the University of Delaware.*

With nineteen country versions, OS/2 offers a solid platform for international application development.

International

National Language Support for the Dialog Manager

Karl J. Engvold, Jr.

by Karl J. Engvold, Jr.

The OS/2 1.2 Dialog Manager (provided in the OS/2 Programming Tools and Information) provides multilingual National Language Support (NLS) for OS/2 application development.

Software development has gone through many changes within the last few years. One of the most significant of these is the growing importance of the international market for applications. Developers doing business internationally have an immediate need to develop applications that will run in many different languages. In order to support this development community, systems suppliers like IBM need to create an environment that fosters the development of these applications. NLS in the Dialog Manager (DM) and its Dialog Tag Language Compiler (DTLC) provide exactly this kind of environment.

The OS/2 1.2 DTLC and Dialog Manager provide NLS for the application developer in the form of NLS code page support, as well as Program Integrated Information (PII) translations into many natural languages. PII, formerly known as Machine Readable Information (MRI), is textual information used by an application to communicate to the user. This information must be in the user's natural language. Additionally, this information must be isolated from the application's executable code so that the same executable code can be used for different natural language forms of the PII. This NLS architecture allows developers the flexibility to develop applications that operate in different language environments with minimal development and packaging effort.

DM's predefined PII provides information to application users. This information is presented in the form of messages, application panels, and help panels. Ideally, this information is presented in the user's language. DM currently supports these languages:

Canadian French
Danish
Dutch
English
Finnish
French
German
Italian
Japanese
Korean
Norwegian
Portuguese
Spanish
Swedish
Taiwanese

Although the U.S. version of DM contains only the English language PII, the PIIs for the languages listed above are available to U.S. customers through the IBM Developer Assistance Program.

USING DTLC AND DM

During the development of a DM application, developers define their own PII (messages, application panels, and help panels), which must then be translated into the languages that they want to support. Once translation is completed, the Dialog Tag Language (DTL) source must be compiled and prepared for use by the DM application. The DTLC allows DTL source to be prepared for all the languages listed in the previous section. The NLS function of the DTLC allows developers

to specify the Country Code and Code Page options for the DTL source being compiled. Refer to the OS/2 1.2 *Dialog Tag Language Guide and Reference* for more details on the syntax of DTL messages, application panels, and help panels. This publication also provides more information about NLS in preparing DTL source for use by a DM application.

Once the DTL source for the DM application is translated and compiled, the developer must ensure that the translated version of DM's PII matches the application's PII. The language version of DM's PII can be selected dynamically during invocation of the DM application by setting the DPATH and HELP variables for an OS/2 session.

Figure 1 is an example of how a natural language can be selected for use by a DM application. The command lines could be placed in an OS/2 command file (for example, RunMYapp.CMD). The example selects the French language by setting the DPATH and HELP environment variables to include C:\TOOLKT12\DM\FR (the location of DM's PII) and C:\APPDIR\FR (the location of the application's PII). The DM runtime facility uses the DPATH environment variable to determine where to look for the PII files it needs (messages and application panels), whereas the OS/2 Information Presentation Facility (IPF) uses the HELP environment variable to determine where to look for its PII files (help panels). The IPF is the help managing facility provided in the OS/2 1.2 base system.

MULTILINGUAL SUPPORT

Within each OS/2 session, users can start multiple DM applications, each of which may use a different natural language. Figure 2 shows an example of multiple DM applications running on the Presentation Manager (PM) desktop, all using different languages.

Each DM window in Figure 2 is associated with a DM application (three DM windows = three DM applications in different languages). The fourth window, entitled "Group - Main," shows that the user started from an English OS/2 system. For each of these DM applications, the Portuguese, French, and German languages are selected.

More detailed information on how to load and select a language can be obtained from the IBM OS/2 1.2 *Dialog Manager Guide and Reference.*

Figure 3 shows the layout of DM's PII as it appears on a user's system, as well as how a DM application accesses this PII information.

Each set of DM PII files for a specific language is contained within a subdirectory found under the \TOOLKT12\DM path. This path, plus the language subdirectory name, must be used in selecting the language for a DM application. Figure 4 shows the subdirectory names used for the languages provided by DM.

```
set dpath=C:\TOOLKT12\DM\FR;C:\APPDIR\FR;%dpath%
set help=C:\TOOLKT12\DM\FR;C:\APPDIR\FR;%help%
start /PM MyDMapp.EXE
```

Note: The DPATH and HELP environment variables can be set directly by the DM application code through OS/2 API calls. Selecting the proper DM PII for the natural language in this way is cleaner than selecting them through the OS/2 command file example above.

Figure 1. OS/2 Command File RunMYapp.CMD

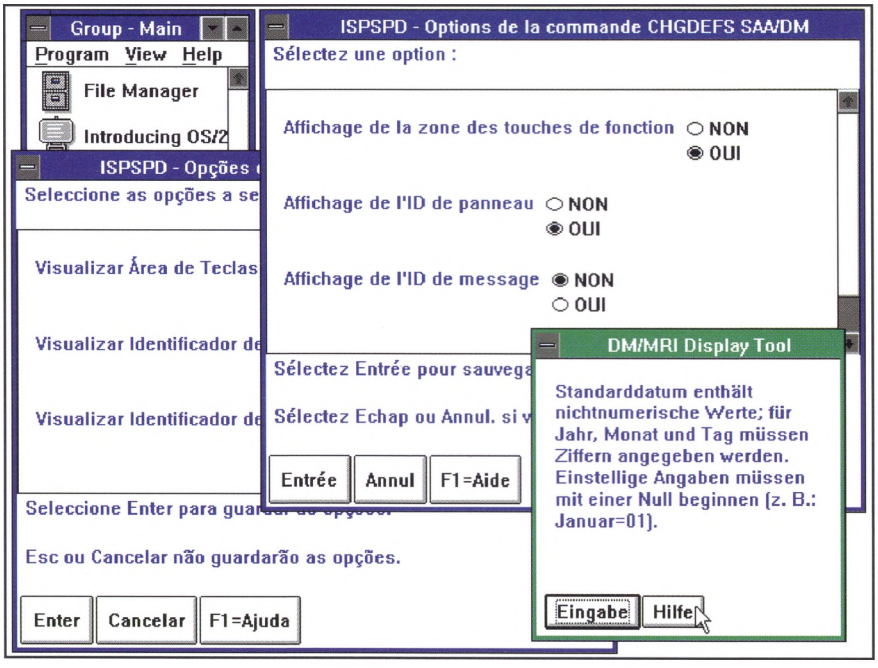


Figure 2. Multiple DM Applications Running on the PM Desktop

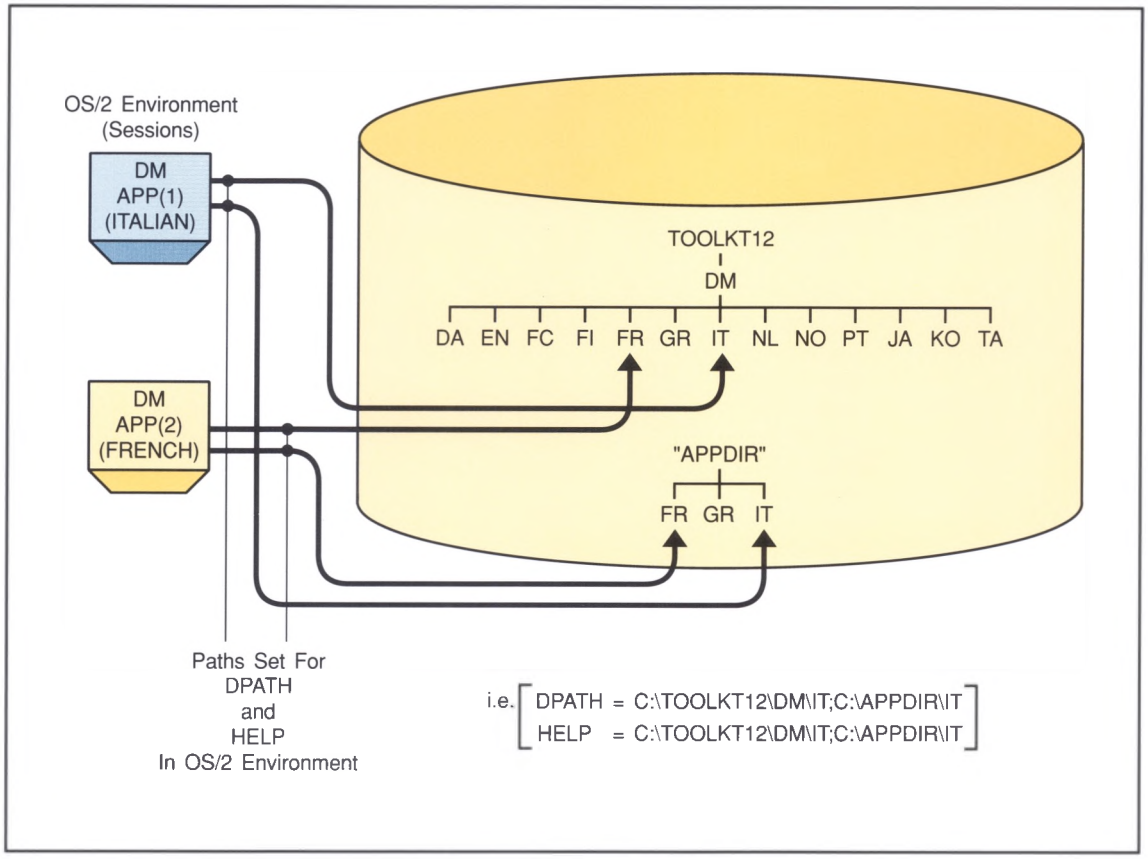


Figure 3. OS/2 Configuration for Selecting a Language in a DM Application

Subdirectory	Language
DA	Danish
EN	English
FC	French Canadian
FI	Finnish
FR	French
GR	German
IT	Italian
JA	Japanese
KO	Korean
NL	Dutch (Netherlands)
NO	Norwegian
PT	Portuguese
SP	Spanish
SV	Swedish
TA	Taiwanese

Figure 4. Subdirectory Names for Languages

This architecture allows a DM application to run in a different natural language environment without requiring any modifications to the application code used to access DM's PII. Therefore, in the multilingual environment, the work for the DM application developer has been made easier!

Karl J. Engvold, Jr., *IBM Programming Systems Laboratory, 11000 Regency Parkway, Cary, NC 27512. Mr. Engvold is a senior associate programmer working in the software development and test areas with products involving computer generated graphics, business graphics, and application development tools. Within his development responsibilities, he specializes in double-byte character set and national language support. Mr. Engvold holds a High School degree from the American School of Paris, France and a BS in Information and Computer Science from the Georgia Institute of Technology.*

Extended Edition Database Manager

Query Manager Customized Interface

by G.E. "Skip" Fisher, Lynda K. Mersiovsky, Alejandra Sanchez-Frank, and Susan L. Watson

The customizing functions of OS/2 Extended Edition Query Manager provide a straightforward, interactive way to develop database applications. This article describes the new features of Version 1.2, which allow developers to create more powerful applications. The example presented shows how, in a few steps, an invoice processing and inventory control system can be implemented.

The OS/2 Extended Edition (OS/2 EE) Query Manager customizing features give developers the ability to create complete database applications. These features include:

- Panels that allow the easy design and layout of screens for data entry and viewing
- Procedures that provide sequencing of commands and control flow
- Queries that permit controlled access to a database
- Menus that allow the creation of user-defined menu screens

In OS/2 EE 1.2, this capability is extended through functionality that allows additional process control:

- Panel actions can be issued by procedures.
- Rules for panels can be specified that associate procedures and queries with panels.

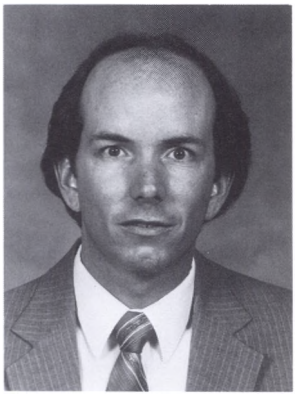
This article focuses on these new capabilities and provides a hands-on example to illustrate their usefulness and power. To introduce the new user to panels, some concepts are discussed including rules-for-panels specification and enhanced panel-procedure interaction. Then, a detailed example shows the actual definition of each Query Manager object (panels, procedures, and queries).

PANEL CONCEPTS

A *panel* can be viewed as an empty page for marking data fields and placing text. This page is a view into data in the database. The panel designer can define functions for searching a database, using a panel as a template for viewing. Data viewed on a panel can be changed or deleted. A panel can also be used to add new data into the database.

The simplest panel is one designed for a single table. The designer can create a single table panel from scratch or have Query Manager build a default panel by simply specifying the table name. The panel designer may also use the default panel as a starting point for designing a customized panel for a single table or multiple tables.

In a panel based on multiple tables, the panel designer must specify the connecting relationships between all the tables on the panel. Only one table can be identified as the root table, which is the top of the hierarchy of tables used. A second table can be identified as the subordinate or subtable to the root table. Database actions, such as updating or inserting rows of data, can only be defined for root and subtables. Other tables can be referenced on the panel as look-up tables. A look-up table is one from which columns can be presented on the panel for viewing only. Information from these tables can also be used for computations.



G.E. "Skip" Fisher

Alejandra Sanchez-Frank

Susan L. Watson

Once a panel has been built, it can run in one of two modes: Add or Change. Add mode inserts new rows into the database. Change mode searches for, changes, deletes, or views existing rows from the database.

Panel instance and *panel set* describe how panels are presented to the end-user when a panel is run. A *panel instance* is a single occurrence of a row in the root table and any tables connected to the root table with matching connecting columns. A *panel set* is one or more panel instances that satisfy a given search condition or search query. Each is sequentially presented to the user.

The user designates which columns in the tables to use for a particular panel by defining fields. A *field* is a variable name given to the data from a particular column of a particular table. Not all fields have unique values within a panel. Fields called *repeated fields* can take on multiple different values. These fields exist for a panel only when the relationship of the root to the subtable has multiple subtable rows connected to a given root row. A scrollable window shows each set of repeated fields. Each repeated row in the window represents one set of unique values for the repeated fields.

Users can define a *computed field*, which contains the results of a computation on one or more fields on the panel. It is defined as a formula containing table field names, arithmetic operators, constants and/or expressions to specify the desired calculation. A formula may also include a function (for example, sum, average) on a repeated field of the panel.

Internally, fields from the panel are defined as locally created variables in the variable pool, with the field name used as the variable name. These variables are available to other objects called from the panel and exist until the object that created them ends. Each panel instance resets all of the panel's local variables when data is read from the database. Any fields that the user changes on the screen are placed in the variable pool whenever the user selects an action. The variable values can also be modified by the Query Manager SET command. The SET command can be embedded in a procedure that is run as the result of an action from the panel.

Panel actions are used during panel execution to manipulate the panel. They are based on a set of panel operations and/or Query Manager commands. The user can identify the operations and commands to tailor the set of actions for the panel. The Query Manager generates all necessary Structure Query Language (SQL) statements to support the specified panel operations. The panel designer chooses which panel actions will be available to the panel user during execution. When a new panel is opened, the panel designer is given a default set of panel actions that can be customized. Some of the panel actions are:

- **ADD AND NEXT**: Performs an addition to the database, adding the row(s) of data entered on the panel.
- **NEXT**: Presents the next panel instance in the panel set.
- **CHANGE AND NEXT**: Performs a change to the database, updating the row(s) changed on the panel.

RULES FOR PANELS

The designer can use the Rules-for-Panels pop-up window to specify the Panel Search Query for the panel. The results of this initial search determine the panel set available (Change mode only) when the panel is displayed for execution.

The designer may also specify the Instance Rules (instance procedures) for the panel (one for Add mode and one for Change mode) from this window. The specified procedure runs automatically before displaying each panel instance. It can access or modify variables for the panel instance. Procedure (Add) specifies the procedure to be called just before each instance is displayed when a panel is running in Add mode. Procedure (Change) specifies the procedure to be called after each panel instance has been read from the database and before each instance is displayed when a panel is running in Change mode.

The End Rule controls the processing when the end of a panel set has been reached. The designer may specify whether to set up the panel for a new search condition or to exit the panel.

PANEL ACTIONS RUN FROM PROCEDURES

Most panel operations are valid as commands in a procedure run from a panel. A procedure runs as the result of a panel action or as an instance procedure. Figure 1 shows the panel operations that can be executed from a procedure and the command syntax.

The Message option determines whether to display informational messages that result from successful execution of Add and Next, Add and Keep, Delete and Next, Change and Next, and Next operations from a procedure.

In addition, a new panel operation called Quit Panel can be executed as a command from a procedure only, and causes panel execution to terminate.

Instance procedures can run the same panel operations as any other procedure run from a panel. When specified, instance procedures run automatically before displaying each panel instance, and can access variables from the panel instance. A looping effect is created when an instance procedure contains an Add and Next, Add and Keep, Delete and Next, Change and Next, or Next panel operation. This is because the operation containing the Next reexecutes the instance procedure for the Next panel instance.

For Change-mode processing, a Delete and Next, Change and Next, or Next operation causes the Next instance in the panel set to be fetched; the instance procedure is reexecuted automatically. The looping continues until the end of the panel set is reached, or the procedure manually terminates execution based on some condition(s). This is used to create "batch" procedures or to bypass processing of specific rows.

For Add-mode processing, the Add and Next or Add and Keep operation effectively causes an "infinite loop" when not followed by Quit Panel because there is no concept of panel set. In this case, the procedure should contain logic to manually terminate execution (for example, by using a counter that is a global variable).

The following section illustrates the process of defining an order-entry and inventory application, using the functions described above.

APPLICATION EXAMPLE

In the application example, a data entry panel captures user input, such as customer's name, address, items ordered, and quantity. Once this is entered, an invoice with appropriate invoice number, unit prices, total prices, and total invoice amount is printed. Inventory is updated by reducing the number of items in stock by the amount ordered.

Panel Operation	Command Syntax in Procedures
Add and Next	ADD (MESSAGE = YES|NO)
Add and Keep	ADD (TYPE = CONT, MESSAGE = YES|NO)
Delete and Next	DELETE (MESSAGE = YES|NO)
Change and Next	CHANGE (MESSAGE = YES|NO)
Next	NEXT (MESSAGE = YES|NO)
Print	PRINT PANEL (TYPE = INSTANCE)
Compute	COMPUTE
Search	SEARCH (TYPE = FIELD SEARCH)
Extended Search	SEARCH (TYPE = EXTENDED)

Figure 1. Syntax of Panel Operations in Procedures

To define the data entry panel, first define the four tables shown in Figure 2.

Figure 3 shows the INVENTORY table, which consists of three columns:

- ITEM contains the name of the item.
- PRICE contains the price of the item.
- #_IN_STOCK contains the current count of the item in inventory.

The INVOICE table (Figure 4) contains a list of invoice numbers associated with the customer, the customer name, and address.

The ITEMS table (Figure 5) shows a list of items ordered, the quantity for each item ordered, and the corresponding invoice number.

TABLE	Column Name	Type	Length	Attributes
INVENTORY	ITEM PRICE #_IN_STOCK	Character (Fixed) Decimal Small Integer	20 5,2	Data Required, Text Data Required Data Required
INVOICE	INVOICE# CUSTOMER_NAME SHIPPING_ADDRESS	Small Integer Character (Fixed) Character (Fixed)	 20 40	Data Required Text Text
ITEMS	ITEM QUANTITY INVOICE_NUMBER	Character (Fixed) Small Integer Small Integer	20	Data Required, Text Data Required
INVOICE #	SAVED_INVOICE_#	Small Integer		Data Required

Figure 2. Table Definitions for the Application Example

INVOICE# table (Figure 6) holds the invoice number for the next order to be processed in column SAVED_INVOICE_#.

After defining the tables, go to the Panels section of Query Manager to define the data entry panel. The result should be a panel with the fields shown in Figure 7. We want the invoice number to automatically appear incremented for each new invoice. We also want the panel user to type the customer name, shipping address, name of the items ordered, and quantity ordered; that is, these should be input fields. Finally, we want the user to be able to see, on request, the unit price for each of the items, the total price, tax, subtotal, and total amounts. Since these are values to be retrieved from a look-up table or to be computed, they should be output fields.

To mark the fields on the screen, start by specifying the panel tables through Table Selections. We declare our INVOICE table as the Root table; the ITEMS table as the Sub-table; and the INVENTORY table as the Lookup1 table:

```
Table Type      Table Name
Root            INVOICE
Sub             ITEMS
Lookup1         INVENTORY
```

Now go to Table Fields, Figure 8, to enter the fields relevant to the application.

Choose a name for each field, and enter it into the Field Name column. Then, specify the type of table from or to which data is to be retrieved or stored. (In the example, the table type can be Root, Sub, or Lookup1 because of the entries that we made in Table Selections.) For each field, specify the table column associated with the field. This column must be a column in the table specified for the table type. Finally, specify the usage for the field; that is, for Search (S), Add (A) or Change (C) modes, or as a Key (K). Search mode is specified for Root table fields only. Fields used as connecting columns from the Root to the Subtable may not have a Change (C) usage. At least one field from the subtable must be defined as Key. These fields uniquely identify a single row in the subtable. We do not specify a usage for Look-up table fields. The Width and Edit Code, if unspecified, default to the width and edit code of the corresponding table column. The width specifies the length of the field and the edit code controls the way the data is displayed. In this case, the fields are numeric

ITEM	PRICE	#_IN_STOCK
Standard Planner	9.99	130
Management Planner	19.99	125
Personal Planner	14.99	143

Figure 3. Inventory Table

INVOICE#	CUSTOMER_NAME	SHIPPING_ADDRESS
100	John Wagner	1234 Madison Ave., Seattle WA 98195
101	Sally Stanford	11400 Burnet Rd., Austin TX 78758

Figure 4. Invoice Table

ITEM	QUANTITY	INVOICE_NUMBER
Standard Planner	8	100
Management Planner	5	101
Personal Planner	10	101

Figure 5. Items Table

SAVED_INVOICE_#
102

Figure 6. Invoice Number Table

(L), character (C), or numeric with two decimal places (L2).

To specify the connecting relationship between all the tables accessed by the panel, select Connecting Columns as shown in Figure 9.

The connection between two tables consists of one column from each table. These columns must contain data values that match. Thus, in the example, connect the two tables using the INVOICE# column from the Root table and the INVOICE_NUMBER column from the Subtable. The connection type is One-to-Many (OM) because each invoice can have more than one item. Connect the Subtable to the Lookup1 table using the ITEM column from each. There is no connection type to be specified when look-up tables are involved.

To define the expressions for the total price of each item ordered, the subtotal, sales tax, shipping costs, and total invoice amount, go to Computed Fields (Figure 10).

The user should be able to select one of two options from the Actions pull-down menu: Compute Total, which displays the unit prices for the items, and the corresponding totals, tax, and total invoice amount; or Submit Order, which updates the inventory and prints the customer's invoice. We can create this menu of options by going to Panel Actions. Figure 11 shows the result of erasing the default panel actions, and specifying Compute Total and Submit Order instead.

The action keys specified provide the user with a faster way to execute the action. The panel operation/command specified is the operation or command that executes each time the user selects this action. If the user selects Submit Order (or presses Ctrl+F2), the procedure SUBMIT_ORDER is run in Add mode (notice the A under Mode in the figure). If the user selects Compute Total (or presses Shift+F10), all the computed fields are calculated and their values displayed.

The correct invoice number should be displayed each time a new order is processed. To do this, we use one of the new features of Panels: Rules for Panels. By selecting Rules for Panels, we define an instance procedure, GET_INVOICE# (Add mode), to be executed each time a new instance of the panel is brought up; that is, GET_INVOICE# will be executed each time the user processes a new order. The correct invoice number is then displayed:

```
Instance Rules
   Procedure (Add)......  GET_INVOICE#
   Procedure (Change).....
End Rule.....................  Search
```

At this point, we mark the fields and position them in the panel's screen as shown in Figure 12. This involves moving and sizing text, fields, and repeated fields by selecting Mark Text or Mark Field from the Layout

pull-down menu. Text and fields become tokens that can be resized easily and dragged around on the screen using the mouse pointer or keyboard cursor keys. This completes the definition of the INVOICE panel.

Now, we must define the procedure, SUBMIT_ORDER, that takes the input values from the user and uses them to add data rows to the INVOICE and ITEMS tables, update the INVENTORY table, and print the invoice as shown in Figure 13.

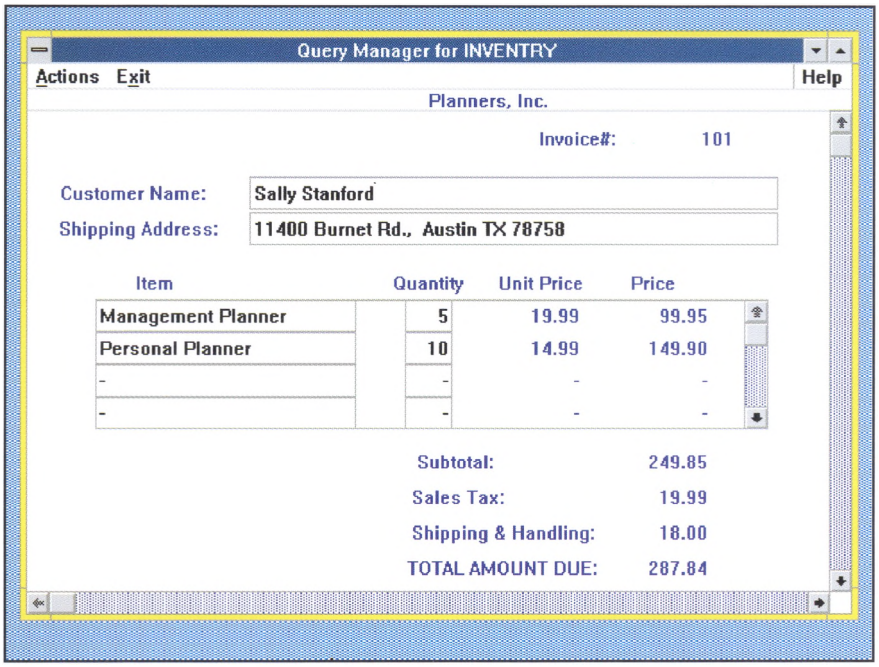


Figure 7. INVOICE Panel

The procedure SUBMIT_ORDER causes the INVOICE panel to add the invoice input values into the database (by using the panel action Add). To update the inventory, it gets each of the invoice items and their quantities, which are stored in the ITEMS table. This is done by running a panel GET_ITEMS that processes each item on the invoice. After processing is complete, it prints the invoice and updates the INVOICE# table by running the query UPDATE_INVOICE:

```
UPDATE invoice# SET saved_invoice_# =
saved_invoice_# + 1
```

The panel GET_ITEMS is a simple panel that is invisible to the user. It has a single table called ITEMS that is defined using Default Definition. We change the table field names that will be used in other objects to unique names. Therefore, we change FIELD1 to ORDER and FIELD2 to AMOUNT in the Table Fields pop-up window. To finish defining this panel, we only need to specify the Rules for Panels:

```
Begin Rule
  Panel search query... GET_ITEMS
Instance Rules
  Procedure (Add).....
  Procedure (Change)... PLACE_ORDER
End Rule.............  Exit the panel
```

The Panel search query, GET_ITEMS, is defined as:

```
SELECT * FROM items WHERE
invoice_number = &invoice
```

Field Name	Table Type	Column Name	Usage	Width	Edit Code
INVOICE	Root	INVOICE#	SA	6	L
CUST_NAM	Root	CUSTOMER_NAME	SAC	20	C
ITEM_ORD	Sub	ITEM	KAC	20	C
QUAN	Sub	QUANTITY	AC	6	L
INV	Sub	INVOICE_NUMBER	KA	6	L
SHIPPING	Root	SHIPPING_ADDRESS	SAC	40	C
ITEM_LU	Lookup1	ITEM		20	C
UNIT_PRC	Lookup1	PRICE		5	L2

Figure 8. Table Fields

From Table		To Table		Connection
Type	**Column Name**	**Type**	**Column Name**	**Type**
Root	INVOICE#	Sub	INVOICE_NUMBER	OM
Sub	ITEM	Lookup1	ITEM	

Figure 9. Connecting Columns

Field Name	Formula	Width	Edit Code
PRICE	UNIT_PRC * QUAN	7	L2
SUBTOTAL	SUM(PRICE)	7	L2
SALESTAX	SUBTOTAL * .08	7	L2
SHIPCOST	SUM(QUAN) * 1.2	7	L2
TOTAL	SUBTOTAL + SALESTAX + SHIPCOST	7	L2

Figure 10. Computed Fields

Action Text	Mnemonic	Action Key	Panel Operation/Command	Mode
Submit Order	S	Ctrl+F2	run proc submit_order	A
Compute Total	C	Shift+F10	compute	

Figure 11. Panel Actions

The panel gets all the items that were ordered in the current invoice. The current invoice number was defined as a table field variable, INVOICE, on the INVOICE panel. The procedure PLACE_ORDER that updates the inventory is run for each instance. The End Rule specifies that, when no more items are to be processed, we exit the panel (that is, return to the procedure SUBMIT_ORDER). See Figure 14.

Now, run a panel that searches a table for the rows satisfying a query. Then, run a procedure for each instance of the panel. In this case, the panel being run is the INVENTORY panel; the table is the INVENTORY table; the instance procedure is UPDATE_INVENTORY (Change mode); and the query is GET_INVENTORY:

```
SELECT * FROM inventory WHERE
item='&order'
```

ORDER is a table field variable on the GET_ITEMS panel.

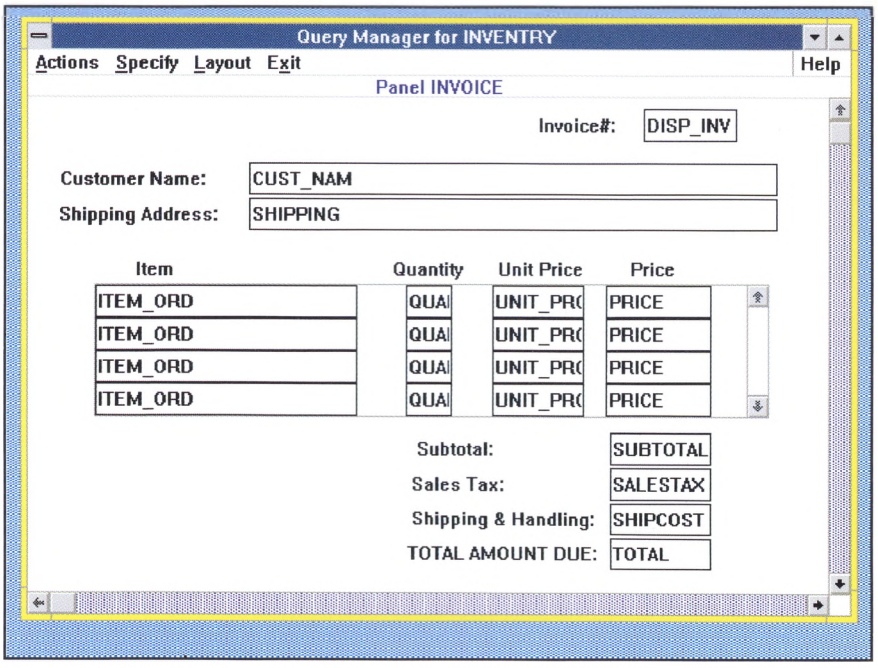


Figure 12. Marked Fields in INVOICE Panel

We define the INVENTORY panel as we did the GET_ITEMS panel. The panel uses the default definition for the INVENTORY table. We change FIELD3 to #STOCK in the Table Fields pop-up window. The rules for this panel are:

```
Begin Rule
  Panel search query.... GET_INVENTORY
  Instance Rules
Procedure (Add).......
Procedure (Change).. UPDATE_INVENTORY
End Rule.............  Exit the panel
```

The instance procedure UPDATE_INVENTORY (Figure 15) subtracts the amount ordered from the amount available in stock, and puts the result back into the table. The GET and SET commands are required to define explicitly the variables used in the procedure. This is the end of the chain.

Returning to the main panel INVOICE, we have defined all of its components except the instance procedure, GET_INVOICE#. We specified this procedure as its instance procedure in Rules for Panels. Procedure GET_INVOICE# only needs to access the data in the table INVOICE#. This is done by way of a panel that gets the data through a query. We call both the panel and the query GET_INVOICE#:

```
SELECT * FROM invoice#
```

Thus, panel GET_INVOICE# uses table INVOICE# in a default definition, with FIELD1 changed to SAVED_I#. The Rules for Panels specifies query GET_INVOICE# as its search query, procedure COPY_INVOICE# as its instance procedure (Change), and Exit the Panel as the End Rule. The two procedures are shown in Figure 16.

We are ready to run the application. It can be run from the command line using "run panel INVOICE (mode=add)," or from the Panel Names window.

This example has been kept simple for illustration. There are many options for expansion that can be added to the application. Some of these are discussed below.

```
/*This procedure is run when the invoice is submitted*/
       'add (message=no)'
       'run panel get_items (mode=change)'
       'print panel (type=instance)'
       'run query update_invoice'
```

Figure 13. Procedure SUBMIT_ORDER

```
/*  This procedure is run for each item ordered  */
       'run panel inventory (mode=change)'
       'next (message=no)'
```

Figure 14. Procedure PLACE_ORDER

```
/*  This procedure is run for each item in an invoice */
       'get current (var1=#stock)'
       'get current (var2=amount)'
       'var1 = var1 - var2'
       'set current (#stock=var1)'
       'change (message=no)'
       'quit panel'
```

Figure 15. Procedure UPDATE_INVENTORY

EXPANDING THE EXAMPLE

Adding or changing procedures and panels can enhance the application and extend functionality. For example, when there is not enough of an item in stock to fill an order, the invoice processing can be halted and a message sent to the sales clerk. A procedure can be included to reorder items with insufficient stock. The items on the order can be marked as back-ordered. Or a panel action can be added to the INVOICE panel that would call a procedure to get the customer's address, and automatically complete this portion of the invoice before submitting the order. Also, menus can be created that act as a front-end to the application by providing an easy way to invoke the INVOICE panel or other objects in the application. Error checking can be added to any procedure. Different paths can be taken in a procedure depending on the values of certain panel fields.

CONCLUSIONS

We have described a subset of the multiple possibilities offered by the Panels function of OS/2 Query Manager. These possibilities include the building of customized database applications in a straightforward, interactive way.

```
/*  Procedure GET_INVOICE#                */
/*  run the panel to get the invoice#  */
   'run panel get_invoice# (mode=change)'

/*  Procedure COPY_INVOICE#                                        */
/*  gets the next invoice# and copies it into the invoice panel */
   'get current (var1=saved_i#)'     /* get saved invoice#      */
   'set current (invoice=var1)'      /* copy into invoice panel */
   'quit panel'
```

Figure 16. GET_INVOICE and COPY_INVOICE Procedures

In OS/2 EE 1.2, many manual tasks can now be automated. The ability to sequence actions within a running panel and the ability to undo actions when the sequence does not complete successfully adds to the customization possible with the Query Manager. We also can enhance our previous applications using the following items:

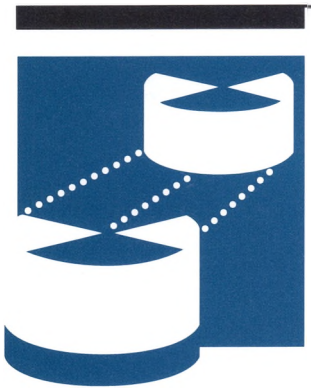
- Validating data automatically
- Printing panel results based on successful conditions as determined by procedure logic
- Bypassing rows that the application does not want the user to process
- Accessing data from tables to be used by a procedure
- Initializing based on procedure logic
- Determining the particular user running the application

G.E. "Skip" Fisher, *IBM Entry Systems Division, Austin Laboratory, 11400 Burnet Road, Austin, TX 78758. Mr. Fisher is a senior associate programmer for the OS/2 EE Query Manager project. He joined IBM in 1987 and has worked primarily on the Panels function with Query Manager. He received a BA in Computer Science from the University of Texas at Austin.*

Lynda K. Mersiovsky, *IBM Entry Systems Division, Austin Laboratory, 11400 Burnet Road, Austin, TX 78758. Ms. Mersiovsky is a senior associate programmer for the OS/2 EE Query Manager project. She joined IBM in 1981 and has worked as a software developer for both DisplayWrite and Query Manager. She received a BS in Computer Science from Texas A & M University.*

Alejandra Sanchez-Frank, *IBM Entry Systems Division, Austin Laboratory, 11400 Burnet Road, Austin, TX 78758. Ms. Sanchez-Frank joined IBM in August 1989 as a senior associate programmer working on OS/2 EE Query Manager. She received an MS in Computer Science from the University of Washington and an MS in Electrical and Computer Engineering from the University of Kansas.*

Susan L. Watson, *IBM Entry Systems Division, 11400 Burnet Road, Austin, TX 78758. Ms. Watson is a development programmer in the Database Management Products Department at the Austin Laboratory. She joined IBM in 1968 in the Federal Systems Division in Houston, Texas, where she worked on NASA contracts for the space program. In 1984, she transferred to the Austin Laboratory and became the development manager for OS/2 Query Manager. She is currently the technical assistant to the database product manager. Ms. Watson received a BA in Mathematics from the University of Texas.*



Extended Edition Database Manager

Database Configuration Parameters

by David C. Baker, William L. Banning, and William W. Myre

This article[1] describes the OS/2 Extended Edition (OS/2 EE) Version 1.1 Database Services and Database Configuration parameters and explains how they can be changed. Database Manager Version 1.2 has some additional parameters and minor changes in the Log parameters. With these exceptions, the parameters described in this article and how they are used continue to apply to the 1.2 release. These interrelated parameters determine the amount of resources allocated by Database Services to handle application requests. They can affect the performance of applications and their ability to run successfully. Developers should consider the impact of changing these parameters when their application shares data or a database with other applications. Performance and concurrency issues become even greater concerns with Database Manager Version 1.2.

The Database Manager has two main areas of configuration: Database Services and the configuration of individual databases. The Database Services Configuration parameters apply to all databases. In addition, each database has its own set of configuration parameters.

Users can set parameters for the application program when installing the application. This may be acceptable if the application is the only application that uses a database, as in the case of turnkey applications. Turnkey applications usually bundle both the hardware and software, and the software provides all the application functions needed. Software to run a dentist's office including such applications as patient scheduling, text processing, billing, and inventory, is an example of turnkey software.

For non-turnkey applications, it may be helpful for developers to document suggested configuration parameter settings and explain how users can change them. This article includes recommendations for parameter settings.

Database Manager Configuration parameters can be modified programmatically, although this may not be justified because the Query Manager (QM) already provides this function. This article describes how to set configuration parameters using the Query Manager. Both the Query Manager menu item name and Database Services Application Programming Interface (API) parameter names are provided. The API name is used in parameter references because it is shorter.

DATABASE SERVICES CONFIGURATION

Database Services Configuration parameters are accessed when Database Services is started (STARTDBM), and are used during

[1] From *Creating Applications with the IBM OS/2 Extended Edition Database Manager* by David Baker, William Banning, and William Myre. Copyright © 1989 by David Baker and William Banning. Published by Addison-Wesley, Route 128, Reading, MA 01867, (617) 944-3700. Suggested retail price $22.95. Reprinted by permission of Addison-Wesley Publishing Company. This article has been edited for publication in the *IBM Personal Systems Developer*.

initialization. The two Database Services Configuration parameters are NUMDB and SQLNSEG.

NUMDB (number of concurrent active databases, range 1–8, default 1) indicates the maximum number of databases that can be open concurrently. The same database can be opened by every OS/2 process. A "START USING database name" request to Database Services causes a database to be opened. Database Services will fail all requests to open an additional database above the number set in NUMDB (will return an error return code).

The value selected for NUMDB affects the value that must be specified for the SQLNSEG parameter. If NUMDB is changed, setting its value to 8 and setting the value of SQLNSEG to 41 should prevent the need to increase these values again as applications are added or as other changes are made to the system.

SQLNSEG (maximum shared memory segments, range 6–802, default 6) limits the number of OS/2 shared memory segments acquired by Database Services. OS/2 shared memory segments are a limited system-wide resource. SQLNSEG is the maximum number of OS/2 shared memory segments that can be acquired before Database Services fails the next distinct database that is opened. The minimum value of SQLNSEG must be greater than or equal to 5*NUMDB+1. Therefore, the minimum value is 6.

As each distinct database is opened, a number of OS/2 shared memory segments are allocated. The number of segments acquired depends on the Database Configuration parameters (discussed in the next section).

Setting SQLNSEG

If the LOCKLIST, BUFFPAGE, or DBHEAP Database Configuration parameters are increased, it may be necessary to increase SQLNSEG. Since this is a limit value, consider setting it high to avoid worrying about it. If this seems wasteful, remember that a fully configured PC might have four fixed disks. If each has a single database and all databases are concurrently used, the minimum value needed for SQLNSEG is 21.

Increasing one of the Database Configuration parameters affecting SQLNSEG to a value of 41, for example, will probably not result in a problem. Note that 41 is less than 1% of the OS/2 shared memory segments.

It is unnecessary to change SQLNSEG unless NUMDB is increased or one of the related Database Configuration parameters is changed. If one of these is changed, it is best to increase NUMDB to a value of 8, thereby preventing the need for any further changes.

To be more precise in determining SQLNSEG, follow these steps:

1. Determine the values for the LOCKLIST, BUFFPAGE, and DBHEAP parameters for each database. Do this by selecting Tools on the action bar of Query Manager Databases Main Menu. Select the Reconfigure Database option from the pull-down menu to display the current settings of the LOCKLIST, BUFFPAGE, and DBHEAP parameters for that database. Make the following calculation for SEGS:

   SEGS = 2 + DBHEAP + ((LOCKLIST + 15)/16) + ((BUFFPAGE + 15)/16)

2. Total the values computed for SEGS (one per database), and add 1 to that; then use this value for SQLNSEG. It is only necessary to compute SEGS for those databases that are used concurrently.

To modify the Database Services Configuration parameters using the Query Manager, start the Query Manager and select System on the action bar of the Query Manager Databases Main Menu. A pull-down menu appears that contains two options used to alter the Database Services Configuration parameters: Reconfigure System and Restore System Defaults.

Use the Reconfigure System option to set the Database Services Configuration parameters. Since the Query Manager does not check input parameters for consistency with respect to the parameter interdependencies, it tries whatever is entered. Query Manager then reports all errors from Database Services.

Database Configuration parameters govern the resource use for individual databases.

The Restore System Defaults option restores the Database Services Configuration parameter default values as provided by IBM.

The Query Manager attempts to stop and restart Database Services so that changes made to the parameters will be put into effect. If another OS/2 task is currently using a database, Query Manager cannot stop the Database Manager because this would probably disrupt other programs. In this case, manually issue a STOPDBM command when the other OS/2 tasks have completed execution. Then issue a STARTDBM command. Database Services prevents any subsequent database connections once its configuration file has been modified. The parameters will not be used until STOPDBM and STARTDBM have occurred.

INDIVIDUAL DATABASE CONFIGURATION

The Database Configuration parameters govern the resource use for individual databases when they are opened. These parameters belong to one of four areas of Database Services, as shown in Figure 1. *Shared memory* indicates storage in an OS/2 shared memory segment. This information is not stored in the database (as are the database's system tables).

Lock Manager Parameters

The following two Lock Manager parameters deal with concurrency between applications using the same tables in a database.

LOCKLIST (range 4–250, default 8) indicates the number of 4 KB pages allocated to hold the lock list. This parameter should be large enough to allow all application transaction workloads to acquire their locks. If there is insufficient lock space, the application receives a -912 SQL error code (out of lock space) and will be rolled back.

Concurrency and the number of locks needed are extremely difficult to predict. LOCKLIST entries exist only for the duration of a transaction and should not tie up storage for very

Lock Manager	
LOCKLIST	Maximum storage for lock list shared memory (in 4 KB pages)
DLCHKTIME	Time interval for checking deadlock (seconds)
Buffer Pool Manager	
BUFFPAGE	Buffer pool size shared memory (in 4 KB pages)
MAXFILOP	Maximum number of database files open per application
MAXTOTFILOP	Maximum number of total files open per application
Log Manager	
LOGFILE	Initial log file size (in 4 KB pages) of disk storage
LOGEXT	Size of one log file extent (in 4 KB pages)
LOGMAXEXT	Maximum number of extents to the log
SOFTMAX	Number of log records to write before taking a soft checkpoint
Storage Manager	
MAXAPPLS	Maximum number of active applications using this database
APPLHEAPSZ	Default application heap (in segments)
DBHEAP	Database heap (in segments) shared memory
SORTHEAP	Sort list heap (in segments)

Figure 1. Individual Database Configuration Parameters

long. The default value should be adequate for most applications. If necessary, increase the default setting by 2 until there are no more -912 error codes. If -912 errors persist, attempt to determine the applications being run. They may not be COMMITing their transactions often enough. This application's usage of the database can also lead to other transactions being locked out for long periods of time.

Pages are suballocated in OS/2 shared memory segments, and thus affect the SQLNSEG parameter. If any changes are made to the LOCKLIST value, review the "Setting SQLNSEG" section in this article to determine whether other changes are required.

DLCHKTIME (range 1–600 seconds, default 10) indicates the interval of time between checks for deadlocks in the set of locks for which an application is waiting. A deadlock condition is undesirable because the work performed by an application before it entered the wait is rolled back, and presumably must be repeated. Many applications put this burden back on the user to repeat commands and input data. Changes to this parameter have no impact on preventing deadlocks.

A deadlock condition can be determined sooner by changing this parameter. The time interval may be reduced for a well-behaved application that keeps its own journal and can automatically recover from deadlock. In an environment with little or no concurrency, set DLCHKTIME high. In either case, the overhead in checking does not significantly affect performance.

Buffer Pool Manager Parameters

The following Buffer Pool Manager parameters deal with the interfaces between the OS/2 File System and Database Services.

BUFFPAGE (range 2–1500 in 4 KB pages, default 16) is probably the most significant parameter with respect to performance. It indicates the number of 4 KB pages used by Database Services to hold disk data for this database. The buffer pool formed by these pages is not shared with any other database. The pool is separate from the buffer pool provided by OS/2 via the BUFFERS parameter in CONFIG.SYS. Data is written in a manner that prevents it from being saved in the OS/2 buffer pool, thus avoiding duplicate caching (saving the data in two different buffer pools).

Make sure the value for BUFFPAGE is at least double the value of MAXAPPLS. The buffer pool pages are suballocated in OS/2 shared memory segments and thus affect the SQLNSEG parameter. If this value is changed, review the "Setting SQLNSEG" section to determine whether other changes are required.

The default is low for most uses. The best way to judge whether this parameter is set correctly is to devise a repeatable test measuring the time it takes to perform some set of tasks. By increasing (or decreasing) the BUFFPAGE parameter and measuring the effect, the best setting for the environment can be determined. Most buffer pools provide improved performance by reducing the need to do disk I/O to reread data previously read. An infinite buffer pool does not help performance of applications that only cause data to be written to the disk.

MAXFILOP (range 2–235, default 20) limits the number of OS/2 files concurrently open for use by Database Services running on an application's OS/2 process thread. It is normally set to a value less than MAXTOTFILOP to allow the application to open its own files while accessing the database.

If the Buffer Pool Manager reaches the OS/2 limit (MAXTOTFILOP, unless the application changed it after starting the database), some open database files must be closed in order to open new ones. Opening and closing files by the Buffer Pool Manager is overhead that reduces performance. Occasional activity of this kind is normal, but opening and closing files whenever a new file is accessed is excessive. Depending on the number of files opened by an application, increasing this value may help performance.

Buffer Pool Manager parameters deal with the interfaces between the OS/2 File System and Database Services.

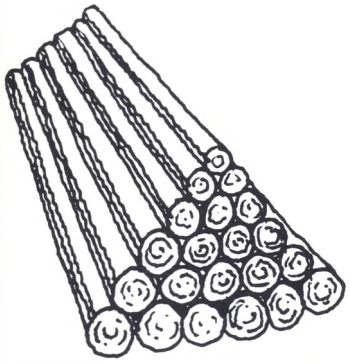
MAXTOTFILOP (range 25–255, default 255) limits the total number of files that can be opened by both the Buffer Pool Manager and the application in a single OS/2 process. When an application starts a database, the current value of MAXTOTFILOP is sent by the Buffer Pool Manager to OS/2 on a SetMaxFiles call. This resets the previous open file limit that OS/2 enforced for that application (OS/2 process). In this way, the Buffer Pool Manager ensures that files can be open to the application and itself. The default is equal to the maximum; little is gained by reducing it.

Log Manager Parameters

Log Manager parameters handle the allocation of disk space for the transaction log.

Log Manager parameters handle the allocation of disk space for the transaction log. It keeps a journal that can be used to back out changes made, but not committed to the database. The primary consideration is that the log can grow large enough to handle a demanding sequence of overlapping transactions without running out of space.

The total size of the log is dictated by the first three parameters that follow. When the log needs to exceed the minimum size (LOGFILE), an extent is acquired if the value of LOGMAXEXT is not zero. In this case, the first log extent (LOGEXT) is allocated and formatted. Then, if the log grows to need more space than the first extent provides, a second extent is allocated and formatted. Assuming space is available from the disk, Database Services continues to allocate extents until it reaches the maximum number of extents (LOGMAXEXT).



If the log becomes full, the operation needing more log space is failed. In this case, that transaction can retry the operation, COMMIT, or ROLLBACK. The latter two actions will free some log space and may allow other transactions to continue. When the log becomes empty (due to transaction inactivity), the LOGEXT areas are released.

There is a performance impact to acquire, format, and free log extents. Therefore, the LOGFILE parameter should be large enough to accommodate transaction peak loads to avoid this overhead. Long running transactions use up log space and may cause extents to be taken. It may be useful to increase the size of the log or the size of the log extents for long running applications that do not COMMIT their work very often. The Import utility is a long running transaction that does a COMMIT only when it completes or runs out of log space. Increasing the size of the log or having log extents for Import is probably not a good idea since it COMMITs when the log gets full.

LOGFILE (range 12–9,999 in 4 KB pages, default 50) determines the minimum amount of disk space to be used for the database's recovery log. This amount of space is allocated when the database is created so that the applications using it are guaranteed a recovery log of at least this size. This parameter, in conjunction with LOGEXT and LOGMAXEXT, determines the total amount of recovery log space available.

This value can be set as high as 4,194 MB using the Database Services API. This is also the maximum size of the log file, irrespective of the values for LOGEXT and LOGMAXEXT. This is considerably larger than the technology provided by most fixed disks, but optical disks (CDROM technology) extend the capacity of storage media by a considerable amount.

LOGEXT (range 4–256 in 4 KB pages, default 10) governs the size of a log extent requested from OS/2 when more disk space is needed for the log after LOGFILE is full and any preceding log extents are full. When the applications using the database COMMIT their work, an attempt is made to delete the log extents. Therefore, log extents are only allocated and used as necessary.

This value can be set as high as 2,000,000 using the Database Services API; however, the total size of the log cannot exceed 4,194 MB.

LOGMAXEXT (range 0–65535, default 20) defines the maximum number of log file extents that will be acquired before a log full condition occurs.

SOFTMAX (range 0–65,535, default 100) is the number of log writes (individual log records written to the log) that occur between soft checkpoints. A soft checkpoint establishes a place in the log from which to begin recovery, so that the Database Manager can skip the first part of the log. The soft checkpoint is most useful when many overlapping transactions cause the log to grow large.

The overhead of a soft checkpoint is limited to one additional log I/O to the disk (an updated log file header record is forced out to disk). The soft checkpoint information is used only to make a database recovery from the log more efficient.

Storage Manager Parameters

Storage Manager parameters handle the allocation of memory (private or shared) for various components of Database Services. The Storage Manager parameters are MAXAPPLS, APPLHEAPSZ, DBHEAP, and SORTHEAP.

MAXAPPLS (range 1–117, default 3) is the maximum number of applications (separate OS/2 processes) that can use the database concurrently. The value of MAXAPPLS should be increased if several concurrent applications will use the database. The value for MAXAPPLS must not exceed half the value of the BUFFPAGE parameter. If the value of MAXAPPLS is increased, changing this value may require an increase to BUFFPAGE.

APPLHEAPSZ (range 2–20, default 3) is the maximum number of segments of private memory available to Database Services when processing application requests. There is usually no need to modify this parameter. If an application receives a -954 SQL error code (out of application heap space), increase the setting of this parameter.

DBHEAP (range 1–45, default 1) is the number of shared memory segments available to process database requests. There is usually no need to modify this parameter. If an application receives a -956 SQL error code (out of database heap space), increase the value of DBHEAP. These segments are allocated as OS/2 shared memory segments and therefore affect the SQLNSEG parameter. If this value is changed, review the "Setting SQLNSEG" section to determine whether other changes are required.

SORTHEAP (range 1–20 segments, default 2) controls the amount of work space available to Database Services for sorting. The default value supplied should be adequate for most applications. SQL SELECTs (whose result table has many rows) that have an ORDER BY clause may fail (SQL error code is -955) due to insufficient work space needed to sort (ORDER) the result. In this case, the SORTHEAP parameter should be increased.

Using the Query Manager to Modify Configuration Parameters

To modify the Database Configuration parameters using the Query Manager, start the Query Manager and then select Tools on the action bar of the Databases Main Menu. The pull-down menu that appears contains two options. The Reconfigure Database option is used to set the Database Configuration parameters. The Restore Database Defaults option restores the Database Configuration parameters provided with the Database Manager.

The Query Manager Reconfigure Database option is used to set the Database Configuration parameters.

SUMMARY

Developers should set or recommend setting the Database Services Configuration parameters to high values since these values are limits. Resources controlled by the Database Services Configuration parameters are allocated only as needed. The parameters prevent (limit) Database Services from using more than a defined aggregate amount of OS/2 system resources for all simultaneously opened databases.

The Database Configuration parameters determine the amount of resources allocated when a database is initially opened, the amount of additional resources needed during operations, and some resource limit parameters. Their settings can affect performance. They may have to be changed to prevent SQL error conditions due to insufficient resources to run applications. More lock, buffer, log, and storage resources are usually required when concurrent database applications use the same data tables.

Resources controlled by the Database Services Configuration parameters are allocated only as needed.

David Baker, *senior systems programmer, Dell Computer Corporation, 9505 Arboretum Boulevard, Austin, TX 78759-7299. Mr. Baker, formerly with IBM, participated in the OS/2 EE Technical Update Seminars as a lecturer on Database Manager and the IBM OS/2 LAN Server product. He received a BS in Computing Science from Rensselaer Polytechnic Institute.*

William L. Banning, *IBM Entry Systems Division, Austin Laboratory, 11400 Burnet Road, Austin, TX 78758. Mr. Banning is a senior architect for the Query Manager portion of the OS/2 EE Database Manager. He holds an MS in Computer Science and a BS in Mathematical Statistics from the University of Missouri.*

William W. Myre, *IBM Entry System Division, Austin Laboratory, 11400 Burnet Road, Austin, TX 78758. Mr. Myre is a technical planner for the OS/2 EE Database Manager project. He joined IBM in 1981 as a programmer on the IBM 5520 office system in the document distribution area. In 1984 he became part of the initial design team of the OS/2 EE Database Manager and did significant portions of the locking, transaction, and recovery subsystems design. Mr. Myre received an MA in Computer Science from the University of Texas at Austin and a BS in Computer Science and Mathematics from Vanderbilt University.*

Market Update

by Bob Tabke

OS/2 SALES FIGURES

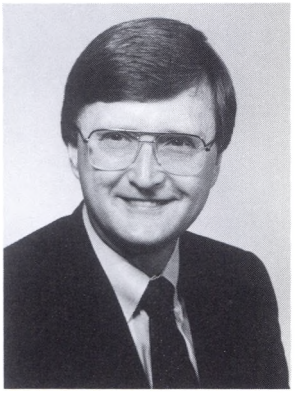
According to one market researcher, OS/2 sales through computer dealers (shown in Figure 1) increased significantly in the last few months. The data reflects estimated sales through computer dealers, excluding direct sales by IBM or Microsoft OEMs. Although it cannot be used to predict overall OS/2 volume, it indicates a significant increase in monthly sales. Factors contributing to the increase are the availability of OS/2 Version 1.2, the availability of more Presentation Manager (PM) applications (see "Market Update" in the *IBM Personal Systems Developer*, Winter 1990), and the declining cost of memory. The high number in December probably reflects year-end buying by large corporations. This microscopic view of OS/2 sales shows that, as the pieces come together, user acceptance increases dramatically.

In predicting long term sales of OS/2, Dataquest bases its numbers on the model established in the past for Microsoft Windows. Figure 2 shows Windows sales history. Dataquest believes that the growth rate for OS/2 will follow a similar pattern, as depicted in Figure 3. According to this pattern, the rate of sales growth is slow for the first two years. After two years, it increases sharply, and continues increasing.

Dataquest is using the Windows model because to date, the environment for Windows in 1985 and OS/2 today evidence some similarities.

- In 1985, the standard hardware platform lacked the horsepower to run Windows effectively. Users were moving to the 6 MHz 80286 IBM PC AT as the high-end machine at that time. Similarly, PC buyers today are moving away from the 80286 toward the 80386-based products needed by OS/2.

- Windows functionality was limited in its first release. Subsequent releases featured improved capabilities and performance. Today, Windows performs well on 80286- and 80386-based machines. New features in each version of OS/2 and the intro-



Bob Tabke

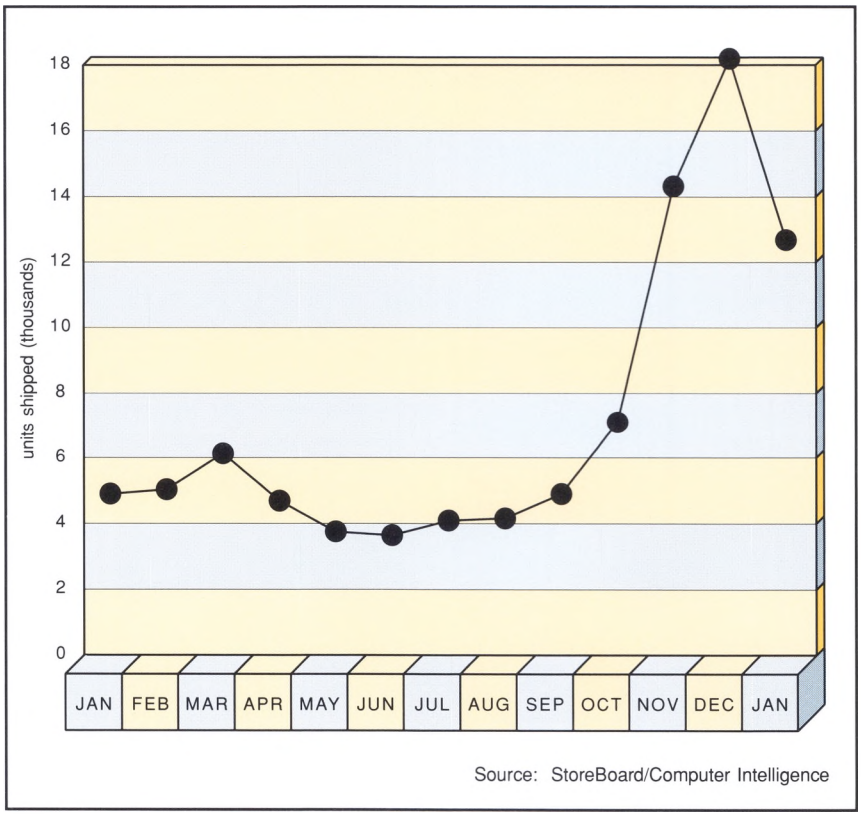


Figure 1. Estimated Sales of OS/2 through Computer Dealers (Past Thirteen Months)

duction of OS/2 2.0 indicate the same evolutionary improvement pattern.

- Windows requires applications designed specifically for it. This means that straight ports are not feasible, and developers had to learn a new style of programming. The OS/2 PM Application Programming Interface (API), with over 800 function calls, is even larger than Windows, and its message-driven architecture is new to most DOS developers. The need to learn a new programming technique has slowed OS/2 PM's acceptance, but with more tools available than for Windows, OS/2 learning will proceed faster.

Acceptance of 386 SX-based systems has been much greater than analysts expected.

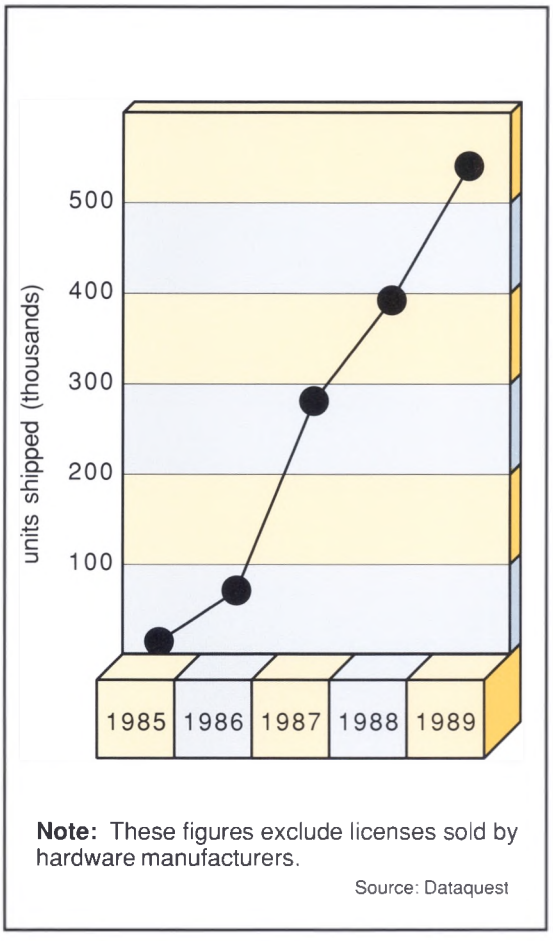


Figure 2. Microsoft Windows Retail Sales

- Microsoft promoted Windows for two and a half years before it achieved significant sales. IBM and Microsoft together are creating a much higher degree of awareness for OS/2.

Dataquest predicts that the "knee" in the OS/2 growth curve (Figure 3) will happen in 1992 when the standard PC hardware configuration will run OS/2. Dataquest sees a low OS/2 sales curve during the next eighteen months, primarily because of the high cost of memory, but also because no "killer" applications are expected (a "killer" application is one that many people want, which can only be provided on OS/2).

Users who are not familiar with OS/2 applications do not know how compelling they can be. This situation could change more quickly than expected. For example, 1-2-3/G, WordPerfect for PM, plus OS/2 2.0 could easily move the "knee" in the curve up into 1991. Previews of these two applications show them to be compelling upgrades to their DOS predecessors. Once users spend time with several applications that truly exploit OS/2, OS/2 sales will accelerate.

HARDWARE TRENDS

Trends in hardware are naturally important to OS/2 application developers. This article describes some of the most recent trends that may affect developers.

386 SX-Based Systems

Acceptance of 386 SX-based systems has been much greater than Dataquest analysts initially expected. Today, it is the basic system for most business desktops. The new 20 MHz 386 SX fills the market currently served by the 20 MHz DX chip, making more computing power available to users at lower prices.

IBM-produced 386 chips are now shipping in some PS/2 models. Supposedly, IBM has rights to modify the design of the 386 that it manufactures. Although it is difficult to predict what affect this may have, it could result in added performance for IBM 386-based machines or lower cost products.

The price for 486-based personal computers may come down faster than expected. While prices of 486-based systems from the major suppliers range from $10,000 to $15,000 today, some aggressive vendors have priced systems as low as $6,000, including monitor.

Dataquest expects to see more 486 suppliers offer systems at the lower end of the price spectrum, in order to gain market share away from 386-based products.

More 286-based machines were sold in 1989 than 386-based machines. Although Dataquest predicts that this will reverse in 1990, inexpensive 286-based clones continue to sell well because their price is $300–$400 less than 386 SX machines. Given the $1,000–$1,800 price range of these machines, this is a significant difference.

The "knee" in the OS/2 growth curve will happen in 1992.

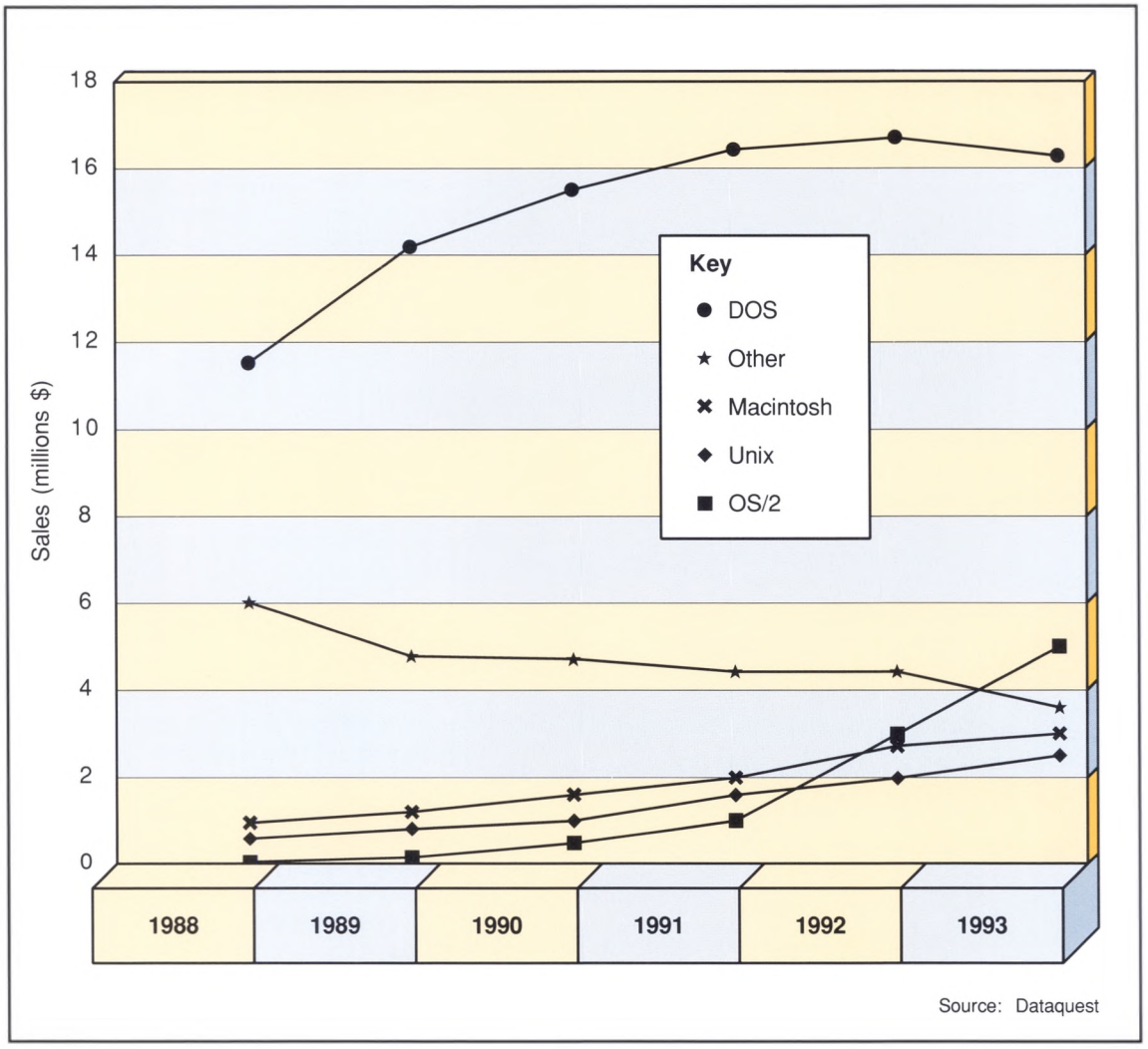


Figure 3. Worldwide PC Operating System Sales

Portables

The portable market continues to grow rapidly, primarily in the notebook segment; growth is slower for 10–15 pound products. A new class of laptops is emerging in the 7–10 pound size, with VGA, 40–100 MB disk, and battery operation. Unfortunately, no breakthroughs in battery technology are on the horizon. Cellular connectivity is a growth area for portables. Depending on the availability of the displays and portable pointing devices, we expect OS/2 to be used on color portables in the next three years.

The 386 SX will probably become the primary processor in most portable systems soon. A low power version of the 20 MHz 386 SX chip is already available to manufacturers.

Realtime is the kind of product that could not exist under DOS.

LOTUS REALTIME TAKES ADVANTAGE OF OS/2

Lotus Realtime is a 1-2-3 Release 3 add-on product that allows securities trading professionals to directly access, analyze, and integrate financial data from online financial services into Release 3 spreadsheets.

Realtime is the kind of product that could not exist under DOS. Realtime works with multiple incoming data streams; it organizes and prioritizes them, and integrates the data into 1-2-3 spreadsheets—all in the background. Lotus Realtime's architecture takes advantage of the OS/2 environment by using OS/2 multitasking to provide true realtime analysis. By allocating analytics and each realtime feed to a separate OS/2 task, Lotus Realtime achieves consistent delivery of data without jeopardizing analytical performance. A Lotus Realtime system has three parts:

- **Lotus 1-2-3 3.0 running under OS/2**: Allows the user to create a customized view of the information and to perform calculations in realtime. To ensure consistency and reliability, Lotus has developed a mechanism whereby the spreadsheet always has the latest price before beginning a recalculation.

- **Lotus Realtime Engine**: Acts as a "traffic cop," sorting the realtime elements pertinent to the spreadsheet, and monitoring up to four datafeeds simultaneously. The Realtime Engine maintains two-way communication with 1-2-3.

- **Feed servers**: Software drivers developed for each realtime data feed using a Realtime Toolkit. A feed server translates incoming digital data (from a source such as a Local Area Network (LAN), satellite, FM subcarrier, or leased line) into a format recognized by the Realtime Engine. It communicates with the Realtime Engine using Named Pipes. Thus, the feed server may be located on another machine and shared by several Realtime Engines. Lotus provides the Toolkit so that third parties can develop feed servers.

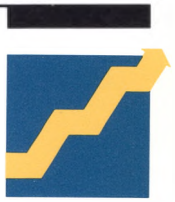
Lotus Realtime is available now and is sold directly by Lotus to trading support professionals and systems integrators. Other possible application areas include data analysis in the laboratory and on the manufacturing floor. The Realtime Toolkit, priced at $25,000, is used to build feed servers. The Realtime Engine is priced between $475 and $1,350 per copy, depending on volume.

OS/2 2.0

This issue of the *IBM Personal Systems Developer* highlights OS/2 2.0. The articles demonstrate the importance of this version from the developer's point of view. The 32-bit version of OS/2 offers users the following advantages over the 16-bit version:

- **Performance**: Applications designed for the 32-bit version run significantly faster.

- **Better DOS**: More DOS memory, up to sixteen DOS sessions, better compatibility with DOS. The availability of over 600 KB of DOS memory, even on a LAN-connected system with Terminate-and-Stay-Resident (TSR) programs, is a compelling benefit to most users. It is particularly important to users of the 65% of business PCs expected to be connected to LANs in 1992.



- **Programming environment**: More capacity, more function, more tools, more portability to other hardware architectures.

Before OS/2 2.0, compelling OS/2 applications (not just DOS ports of Windows applications) were needed to get users to adopt OS/2. OS/2 2.0 itself will be compelling to many users because of the better DOS capability. It is reasonable to expect that many initial sales of OS/2 will be to users wanting the better DOS capability. Once they have OS/2, however, they will try one or two OS/2 applications and will see how good they really are. This is the necessary gradual road that will lead to widespread adoption of OS/2 applications. In the meantime, the operating system itself now justifies its purchase and the memory to support it. This is good news for OS/2 applications developers who have been waiting for signs of growth in the OS/2 market before developing applications.

Why OS/2 Will Succeed

The components that will drive OS/2's success in the 1990s are the very ones that restrained it in the 1980s: hardware pricing and available applications.

Dataquest projects that by 1991 the standard machine will have 4 MB of memory, just as the current memory standard is 1 MB. If a company standardizes on a 4 MB machine, then *users will not be able to buy machines that do not run OS/2.*

OS/2 Presentation Manager applications are finally starting to arrive; the last issue of the *IBM Personal Systems Developer* lists almost three dozen. The development of these products, combined with the OS/2 development in corporate MIS departments, is producing a group of programmers knowledgeable about OS/2. This, combined with the increased number of available applications, encourages more OS/2 development to take place.

Bob Tabke, *Senior Associate, Dataquest, Inc., 1290 Ridder Park Drive, San Jose, CA 95131. (415) 493-4352. Mr. Tabke is a consultant to Dataquest in the personal computer and telecommunications areas. He has a BS in Electrical Engineering from Purdue University and an MBA from Harvard University.*

The last issue of the Developer lists almost three dozen PM applications that are shipping.

OS/2 B·O·O·K·S

A significant number of books have been written about OS/2 to help both developer and user to better understand and utilize the powerful features of the system. The *Developer* thanks Robert Shook, IBM, Austin, Texas for his help in creating this list. The following is a selected list of those books:

Aaron, Bud. *OS/2 Presentation Manager Developer's Guide*. TAB Books, 1989. 480p.

Aaron, Bud. *OS/2 Presentation Manager User's Guide*. TAB Books, 1989. 380p.

Alonso, Robert. *OS/2 Reference Guide*. Scott Foresman, Forthcoming.

Baker, David C.; Banning, William I.; Myre, William W. *Creating Applications with the IBM OS/2 Database Manager*. Addison-Wesley, 1989. 320p.

Beam, J. Emmett. *Illustrated OS/2*. Illustrated Series. Wordware Publishing, 1988. 264p.

Borland, Russell. *Running OS/2*. Microsoft Press, Forthcoming.

Busch, David. *Supercharging OS/2 Batch Files & Utilities*. Addison-Wesley, 1988. 304p.

Campbell, John. *Inside OS/2: The Complete Programmer's Reference*. TAB Books, 1988. 224p.

Conklin, Dick. *OS/2: A Business Perspective*. John Wiley & Sons, 1988. 256p.

Cortesi, David E. *The Programmer's Essential OS/2 Handbook*. M & T Publishing, Inc., 1988. 710p. Book and Diskette.

Crooks, Ted. *Compute!'s Quick & Easy Guide to OS/2*. Compute! Publications, Inc., 1988. 256p.

Duncan, Ray. *Advanced OS/2 Programming: The Microsoft Guide to the OS/2 Kernel for Assembly Language & C Programmers*. OS/2 Programmer's Library Series. Microsoft Press, 1989. 800p.

Duncan, Ray. *Essential OS/2 Functions: Programmer's Quick Reference*. Programmer's Quick Reference Series. Microsoft Press, 1988. 208p.

Edelhart, Mike; Strom, David. *Managing OS/2: Profiting from Changing Standards*. Brady Books, Forthcoming.

Ezzell, Ben. *Writing OS/2 Resident Utilities*. Addison-Wesley, 1989.

Fernandez, Judi; Ashley, Ruth. *The Power of OS/2: A Comprehensive User's Manual*. TAB Books, 1989.

Fosdick, Howard. *Best Book of: OS/2 Database Manager*. Howard W. Sams & Company, 1989. 300p.

Fosdick, Howard. *OS/2 Database Manager: A Developer's Guide*. John Wiley & Sons, 1989.

Holzner, Steven. *OS/2 Assembly Language*. Prentice Hall, Brady Books, 1990. 512p.

Hyman, Michael. *Running OS/2*. Bantam Books, 1988. 256p.

Iacobucci, Ed. *OS/2 Programmer's Guide*, Vol. 1, 2d ed. McGraw Hill, 1990.

Jamsa, Kris. *Using OS/2*. Osborne-McGraw, 1988. 600p.

Jamsa, Kris. *OS/2: The Pocket Reference*. Osborne-McGraw, 1988. 128p.

Jamsa, Kris. *OS/2: The Complete Reference*, 2 vols. Osborne-McGraw, 1988. 1000p.

Jamsa, Kris. *OS/2: Power Programmer's Guide*. Osborne-McGraw, 1989. 1008p.

Jamsa, Kris. *OS/2: The Complete Reference*, 2 vols. McGraw Hill, 1989.

Jamsa, Kris. *OS/2: Power User's Guide*. McGraw Hill, 1989. 1008p.

Jamsa, Kris. *OS/2 API: The Pocket Reference*. Osborne-McGraw, 1988. 128p.

Krantz, Jeffrey I.; Mizell, Ann M.; Williams, Robert L. *OS/2: Features, Functions, and Applications—Standard Edition 1.0*. John Wiley & Sons, 1988. 288p.

Lafore, Robert; Dror, Asael. *Presentation Manager Programming: An Introduction*. McGraw Hill, 1989.

Lafore, Robert; Norton, Peter. *Peter Norton's Inside OS/2*. Brady Books, 1988.

Letwin, Gordon. *Inside OS/2*. Microsoft Press, 1988. 304p.

Levanthal, Lance. *Assembly Language Programming under OS/2*. Bantam Books, 1989. 416p.

McClure, Rhyder. *Fast Access OS/2*. Prentice Hall, Brady Books, Forthcoming.

Microsoft Press Staff. *Microsoft LAN Manager 2.0 Programmer's Reference*. Microsoft Press, Forthcoming.

Microsoft Press Staff. *Microsoft OS/2 Programmer's Reference*. Vol. 4. Microsoft, 1990.

Microsoft Press Staff. *OS/2 Programmer's Technical Reference Series*, 3 vols. Microsoft Press, 1989.

Moskowitz, David; Ivens, Kathy. *OS/2 Power Tools*. Prentice Hall, Brady Books, 1989. 400p. Includes 1 Diskette.

Moseley, Marshall L. *Compute!'s Using Presentation Manager*. Compute! Publications, Inc., 1989. 256p.

Mueller, Scott; Gobel, David. *Using OS/2*. Que Corp., 1989. 800p.

Murray, William H. *OS/2 Presentation Manager Graphics: An Introduction*. McGraw Hill, 1989.

Murray, William H.,III; Pappas, Chris H. *Assembly Language Programming under OS/2*. Osborne-McGraw, 1988. 700p.

Nguyen, Thuyen; Moskal, Robert. *Advanced Programmer's Guide to OS/2*. Brady Books, 1988. 600p.

Norton, Peter. *Peter Norton's OS/2 On-Line Guide*. Productivity Series. Brady Books, 1990.

Paquette, Daniel; Paquette, Annette. *Understanding OS/2 with Presentation Manager*. Howard W. Sams & Company, 1989.

Petzold, Charles. *Programming the OS/2 Presentation Manager: The Microsoft Guide to Writing Applications for OS/2's Graphical Windowing Environment*. Microsoft Press, 1989. 864p.

Petzold, Charles. *OS/2 Graphics Programming Interface*. Microsoft Press, Forthcoming.

Phoenix. *A BIOS for IBM PS/2 Computers & Compatibles: The Complete Guide to ROM-Based System Software for OS/2*. Addison-Wesley, 1989.

Productivity Solutions, Inc. Staff. *Converting Applications to OS/2*. Brady Books, 1988. 300p.

Quedens, Edmund G.; Beason, Pamela S. *Developing Presentation Manager Applications: An Introduction*. Scott Foresman, 1989. 352p.

Reich, David E.; Robinson, Robert F.; Cheatham, Paul W. *OS/2 Presentation Manager Programming*. John Wiley & Sons, 1989. 620p.

Richardson, Ronny. *MS-DOS Batch File Programming including OS/2*. TAB Books, 1988.

Robbins, Judd. *Essential OS/2*. 2d ed. Sybex Books, 1989. 400p.

Ryan, Ralph. *Microsoft LAN Manager: A Programmer's Guide*. Microsoft Press, 1990.

Schildt, Herbert. *OS/2 Presentation Manager Made Easy*. Osborne-McGraw, 1989. 350p.

Schildt, Herbert. *OS/2 Programming: An Introduction*. Osborne-McGraw, 1988. 500p.

Schildt, Herbert. *Using Microsoft C: Covers Microsoft Quick C & OS/2 C*. McGraw Hill, 1989. 550p.

Schmitt, David A. *The OS/2 Programming Environment*. Prentice Hall, 1988. 320p.

Shaw, Richard H. *OS/2 Programming*. Que Corp., 1990. 450p.

Sheldon, Thomas. *Introducing OS/2*. McGraw Hill, 1988. 320p.

Southerton, Alan. *Advanced Presentation Manager Programming: The Graphics Programming Interface*. Addison-Wesley, 1989. 624p.

Southerton, Alan. *Programmer's Guide to the Presentation Manager*. Addison-Wesley, 1989. 752p.

SPSS, Inc. Staff. *SPSS for OS/2: Operations Guide*. SPSS Inc., 1990. 208p.

Taylor, Allen G. *Guide to OS/2*. Scott Foresman, Forthcoming.

Taylor, R.; Kanai, June. *Using OS/2*. Logic Ware Series. M-USA Business Systems, Forthcoming.

The Waite Group Staff. *The Waite Group's OS/2 Programmer's Reference*. Howard W. Sams & Co. 1988. 550p.

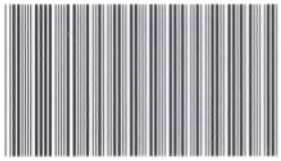
Weinman, Bill. *OS/2 for the Professional Programmer*. MIS:Press, 1988.

Westwater, Raymond. *Writing OS/2 Device Drivers*. Addison-Wesley, 1989. 450p.

Young, Michael J. *Programmer's Guide to OS/2 Presentation Manager*. Sybex Books, 1989. 683p.

Young, Michael J. *Software Tools for OS/2*. Addison-Wesley, 1989.

© IBM Corporation
All Rights Reserved

International Business Machines Corporation
P.O. Box 1328
Boca Raton, FL 33429-1328

G362-0001-05